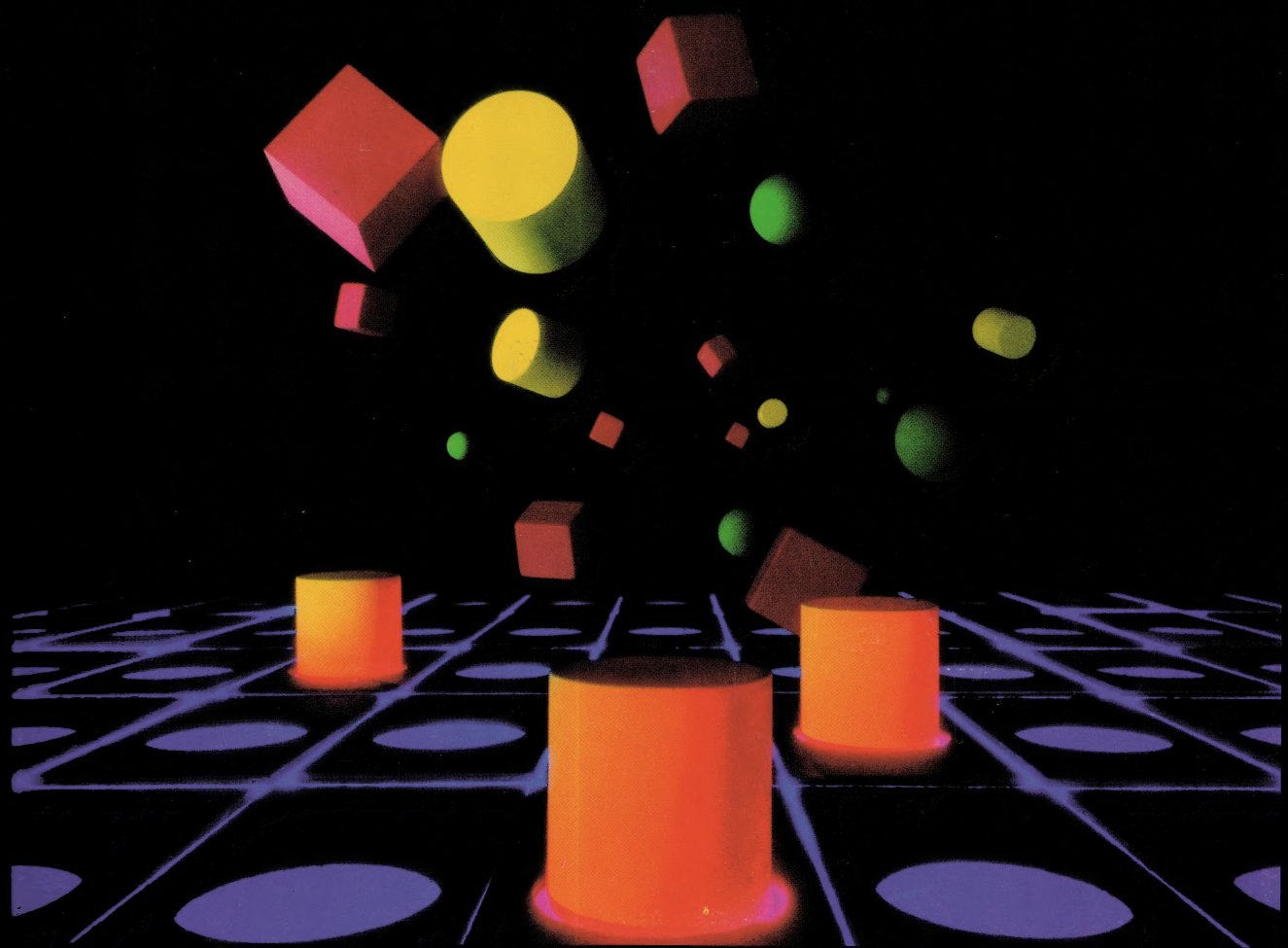
HAYDEN book/software

"A COMPLETE COURSE FOR THE ABSOLUTE BEGINNER"

Apple® Assembly Language Programming

INCLUDES A FULL-FEATURED ASSEMBLER/DISASSEMBLER FOR THE APPLE II (WITH APPLESOFT), II PLUS, IIe, AND IIc

Malcolm Whapshott



Apple®
Assembly Language Programming

Dr. Watson Computer Learning Series

Apple® Assembly Language Programming

Malcolm Whapshott

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

Printed in the United States of America

1 2 3 4 5 6 7 8 9 PRINTING

84 85 86 87 88 89 90 91 92 YEAR

CONTENTS

Apple®
Assembly Language Programming

CHAPTER 1

Getting Started

Understanding Assembly Language

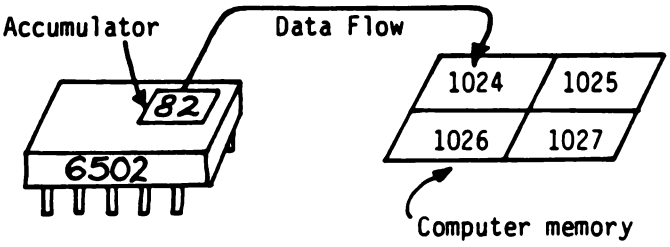
WHEN YOU TURN ON YOUR APPLE, IT MAKES IT QUITE CLEAR THAT the language that it wishes you to speak is BASIC. However, although the language that YOU use is BASIC the only language that the 6502 microprocessor inside your Apple understands is machine code. It's a pretty descriptive name, "machine code", because it is the code which the machine (actually, the 6502) understands. So, when a program in BASIC is run, the control program that is built into the Apple itself translates this into machine code one line at a time as the program is actually running. Naturally, therefore, this process of 'interpreting' the BASIC language takes time - in fact a lot of time - and BASIC, thus runs very much slower than machine code.

Why then don't we mortals program in machine code if it's so much faster? Well, some masochists do but, as a machine-code program is just a list of numbers, it's rather difficult to understand. Fortunately, there is one way that a machine-code program can be written without the pain of direct coding and that is by means of an Assembler. This is a program that translates a program directly into machine code and stores it in memory as a machine-code program. Hence, when this machine-code program is run by the 6502, it runs as fast as any other machine-code program.

When using this technique, the programmer writes an 'assembly language' program which the assembler then operates upon to translate into machine code. Such an assembly language program is very different from a BASIC program as each assembly language command or 'instruction' can be directly translated into a corresponding machine-code instruction, whereas in BASIC, a single command may be actually made up from many machine-code instructions.

Let's take an example to demonstrate the relationship between assembly language and machine code. Let's look at an instruction that takes the contents of one of the 6502's own personal memory locations - called the accumulator - and stores a copy of it at another memory location i.e.

STA 1024



In machine code, this program would read:

100011111 00000000 00000100

and in assembly language:

STA 1024 or

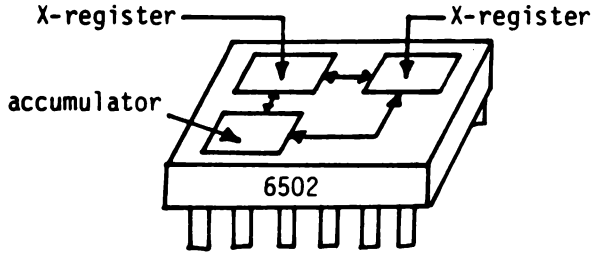
STore the contents of the Accumulator in 1024

Clearly, the assembly language form is much easier to interpret as the instruction 'STA' contains identifiable letters from 'STore the contents of the Accumulator'. The term given to STA is 'Mnemonic' taken from the name of the Greek Goddess 'Mnemosyne', the goddess of memory, a mnemonic being something that is easier to remember.

As far as the programmer is concerned then, the routine is:

- Run the assembler
- Enter the assembly language program
- Assembler translates this into machine code
- Run the machine code program.

The 6502 chip contains three bytes of internal ('on-board') memory, i.e. memory that is actually built into the chip itself. These three bytes are known as the accumulator, the X-register and the Y-register, and are used to store data temporarily while it is being manipulated.



Of the three, the accumulator is the one most frequently used and is really the heart of the 6502. Most of the data flowing through the 6502 passes through the accumulator and most answers to 'sums' end up there.

Many programs entail the loading of data directly into the accumulator, this using what is known as the 'immediate' mode. The mnemonic for this is:

LDA # LoaD the following number into the Accumulator (in immediate mode)

When the 6502 meets the machine-code equivalent of LDA #, it looks for the next character to find what it is required to load into the accumulator. Thus:

```
LDA # 255
```

would put a 255 into the accumulator.

Once the number is in the accumulator, it can be retrieved by means of another instruction that allows a program to store the contents of the accumulator in a particular memory location i.e.

STA STore contents of Accumulator in the address specified.

If this memory location is between 1024 and 2023, then the number taken from the store will be displayed on the screen.

All the STore and LoaD commands you will meet should really be thought of as COPY commands as they create a SECOND copy of that data, leaving the source of this unaltered.

Note that the TRANSFER commands you will meet should really be thought of as COPY commands as they take a copy of the data and create a SECOND copy of that data, leaving the original unaltered.

Let's have a go then at running a machine-code program!

We will create a program which will put a number into the accumulator and then transfer it to the top-left position on the screen, i.e. location 1024.

A couple of points about the assembler: when you start to write a program the assembler needs to be told where you want the program to be placed in the APPLE //e's memory. The APPLE //e has plenty of memory available where we can put a machine-code program (in theory, we could use the whole 64k). However, for short programs page 3, which extends from 768 to 975, provides a convenient 207 bytes. I say convenient as programs which are stored there are unlikely to get mixed up with any BASIC programs. So in our early programs we will make use of page 3 and tell the assembler to start the programs there.

A second point about the assembler is that initially we will only use decimal format for numbers. Later on we shall see how other number formats may be used.

Lastly, the assembler must be told when we have finished entering the program. Thus, the first and last lines of the assembly program are:-

```
START ADDRESS? 768
.................
.... program .....
.................
END
```

N.B. The underlining of START ADDRESS? in the above is intended to indicate that this has been typed by the computer. This convention is used throughout this book and should make it easier for you to interpret the examples. Remember, if it is underlined, then the computer will type this; if it isn't underlined, then this is the bit that you will have to type in yourself.

The first and last lines shown above have nothing to do with the machine code program. They simply provide information to the assembler. (The word END is called a PSEUDO-CODE or PSEUDO OP).

When you have put in your machine-code program you may run it by using the RUN facility provided by the assembler (or by using a call to the start of the machine code program i.e. CALL 768 in this case). Either way you must tell the program to return from machine-code to BASIC or monitor. The command that does this is ReTurn from machine-code Subroutine or RTS.

Right, to put that into the program we must:

1. Tell the assembler that the START ADDRESS? is 768.

2. LoaD number '0' into accumulator A using Immediate Mode. The Mnemonic for this is LDA # followed by the number to be loaded i.e. LDA #0

3. STore in a specific address the contents of the Accumulator, the Mnemonic is STA. After this we must tell the 6502 what the address is i.e. STA 1024

4. ReTurn from the Subroutine to BASIC i.e. RTS

5. Tell the assembler (not the 6502) to END

or

PROGRAM 1.1

```
START ADDRESS? 768
LDA #0
STA 1024
RTS
END
```

Now to enter this:-

a) Load the ASSEMBLER program into your machine.

b) Type in RUN <return>

c) Screen shows MENU

d) Select 'E' to Enter the program. Screen tells you to enter the assembly language program and prompts with START ADDRESS?

e) Type in "768" (without the quotes " of course) and press the <return> key. After the assembler has entered the code, it will type a "?" then:

f) Type in "LDA" <space> "#0" press <return>

g) Type in "STA" <space> "1024" <return>

h) Type in "RTS" <return>

i) Type in "END" but do not press return just yet

At this stage your program should appear as follows:-

```
START ADDRESS? 768
? LDA #0
? STA 1024
? RTS
? END
```

If it does, then O.K. press <return> and carry on to j). If it doesn't, press <return> and then go back to d).

Program returns to MENU.

j) Select "R" to Run the program. Screen asks for address of program start.

k) Type in "768" <return>. After this input the program first clears the screen and then runs the machine-code program which will print an inverse "@" in the top left-hand corner of the screen.

l) Press any key and continue. Program returns to MENU.

Now select 'L' to list the program.

The program then asks you for the START ADDRESS?

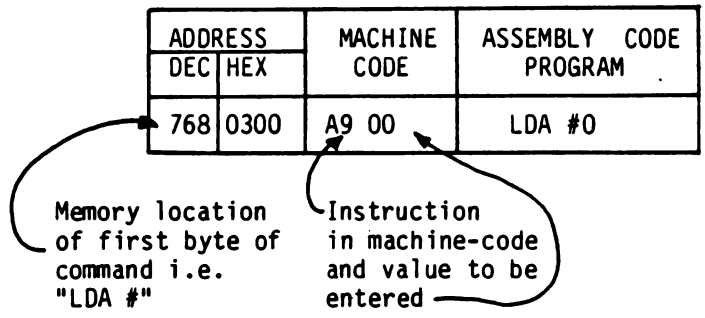
Type in "768" <return>

The screen displays:-

ADDRESS		MACHINE	ASSEMBLY CODE
DEC	HEX	CODE	PROGRAM
768	0300	A9 00	LDA #0
770	0302	8D 00 04	STA 1024
773	0305	60	RTS
etc.		etc.	etc.

Your machine will show a whole screen of data but anything below the line beginning 773 will be ignored as the 6502 reads the RTS (ReTurn from machine-code Sub-routine) and returns to BASIC.

What the above shows is, taking the first line of program 1.1



Thus the listing is both a check on what you entered and also gives you the full machine-code program or the object program. This object code is:

```
A9  00
8D  00  04
60
```

This object code could be entered directly into memory and would yield the same results as the program you typed in. The assembly language only helps you to compile the program in the first place.

Now let us look at another instruction and use this in a program. As stated earlier, the accumulator is the repository of most "answers" and the new instruction ADC "does a sum" and loads the answer into the accumulator.

ADC	ADd with Carry contents of specified memory location to the accumulator.

To do this, however, we must first add two lines to the front of the program. These lines simply get the 6502 ready to do some adding. Don't worry what they mean for now - just type 'em in! - and follow the instructions.

One other point about the jargon! The term INSTRUCTION is used to describe an executable machine code statement. Thus it could consist of LDA # or just RTS. However, the term is also used to refer to the mnemonic alone, as when one says the 6502 instruction set. In this book, the term COMMAND is used to refer to the mnemonic part of an instruction when this precision is required. For instance, in the instruction LDA #, the LDA part may be referred to as the command.

Let's look at the stages:-

PROGRAM 1.2

START ADDRESS? 768	Gives address for beginning of program.
CLD	
CLC	Gets the 6502 ready for adding.
LDA #1	LoaD "1" into the Accumulator in IMmediate mode.
STA 1024	STore the contents of Accumulator (1) in 1024.
LDA #2	LoaD "2" into the Accumulator in IMmediate mode.
ADC 1024	ADd Contents of 1024 (1) to the contents of the accumulator (2).
STA 1026	STore the contents of the Accumulator (3) in 1026.
RTS	ReTurn from machine-code Subroutine.
END	End assembly.

Right then, let's type it in!

If you make a mistake before pressing return, you may correct the mistake using the cursor keys as normal. However, if you press return before you notice the mistake - just type in "END" and start again.

1. Run the Assembler program if not already running.
2. Select "E" to begin entering assembly program.
3. Tell assembler where to start, i.e. type in "768" <return>
4. Type in "CLD" <return> (press <return> after each entry).
5. Type in "CLC"
6. Type in "LDA #1"
7. Type in "STA 1024"
8. Type in "LDA #2"
9. Type in "ADC 1024"
10. Type in "STA 1026"
11. Type in "RTS"
12. Type in "END"
13. Select "R" to run program and then give start address - 768. Screen displays an "A" in 1024 and a "C" in 1026.

Press any key to return to menu.

If you wish to list, select "L" and then give the location "768".

Another way of looking at the two lines

```
LDA #1
STA 1024
```

is as a way of putting a "1" into memory or of printing an APPLE ASCII "1" (an inverse "A") on the screen.

The 6502 has two index registers in addition to its accumulator and these are referred to as Index registers X and Y and each can store one 8-bit number. The arrangement of these or, as the jargon has it, the ARCHITECTURE of the 6502 is shown below (in part) in Fig. 1.1.

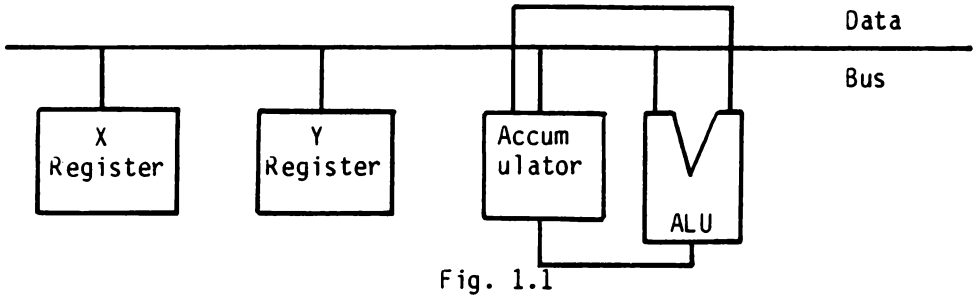


Fig. 1.1

In this figure the X and Y registers are shown identically, although they do differ slightly. Nevertheless, they are both index registers. The real advantage of index registers is that we can increment (increase by 1) or decrement (decrease by 1) the value which they contain and, in addition, we can use them to 'step through' memory, so they are very powerful as we shall see later. To the right of the figure is the 'ALU' or Arithmetic and Logic Unit which is used by the 6502 for all arithmetic and logical operations which it needs to carry out. The ALU has two inputs for the data that it manipulates and one output which feeds the result of the operation into the accumulator. Notice that almost all data flows through the accumulator and this makes the accumulator a key feature of the 6502. Data flows between the various registers along the 'Data Bus' which is a common pathway for communication **within** the 6502. For talking to devices beyond the chip this data bus is extended to access memory also.

In the remainder of this chapter, we will look at these registers and the ways that data can be fed in, out and between these.

First of all we'll have a go at using the X-register - so to load this we use the instruction

LDX	LoaD index register X with data from the specified address, i.e.
LDX 900	means LoaD index register X with the data in memory location 900.

LDX differs from the earlier "LDA #" (apart from one loading the Accumulator and one the X register) in that the LDA # command is an Immediate Mode command. When the 6502 sees this it looks for what's immediately following the instruction and loads that - as data - into the Accumulator. With the new command above "LDX" the 6502 looks for what follows and this specifies the ADDRESS of the data. Thus with the instruction:-

LDX 900

the 6502 goes to memory location 900 to find the data which it loads into the X-register. This instruction (as are all the register instructions) is really a COPY as the data put into the X-register is COPIED from location 900. That is to say, the data originally stored in memory location 900 remains there.

To recover the data we may use the instruction:-

STX	STore the contents of register X in the specified address, i.e.
STX 1024	means STore contents of X register into memory location 1024.

Here's the program!

PROGRAM 1.3

START ADD? 768

```
LDA #1          Load '1' into the accumulator.
STA 1024        Store the contents of accumulator in
                1024.
LDX 1024        Load into X-register, contents of
                memory location 1024 (i.e. "1").
STX 1026        Store contents of X-register in 1026
                (i.e. 1)
RTS             Return from machine-code subroutine.
END             End assembly.
```

At Menu select 'R' to run the program.

The screen should display "A", "space", "A" (both A's coloured white) at the top left-hand position.

By now you should be able to write simple programs so, as an exercise try the following:-

> EXERCISE 1.1
>
> Load the Accumulator directly with a '1', display this (a screen 'A') in 1024. Answer in Chapter 9.

Don't forget to put in the "RTS" at the end. If you do forget then the 6502 will run on to see what it can find and try to execute this. If you are lucky, then the 6502 may simply return to BASIC. However, with your luck, it will probably find something that crashes the system. The crash may be recoverable, in which case pressing the RESET key may return the APPLE to BASIC. If pressing the RESET key does not restore you to BASIC then the crash is not recoverable and it will be necessary to switch the APPLE off and on again, re-load the assembler and start again.

> EXERCISE 1.2
>
> Write your name in the top left hand corner of the screen. One possible answer in Chapter 9.

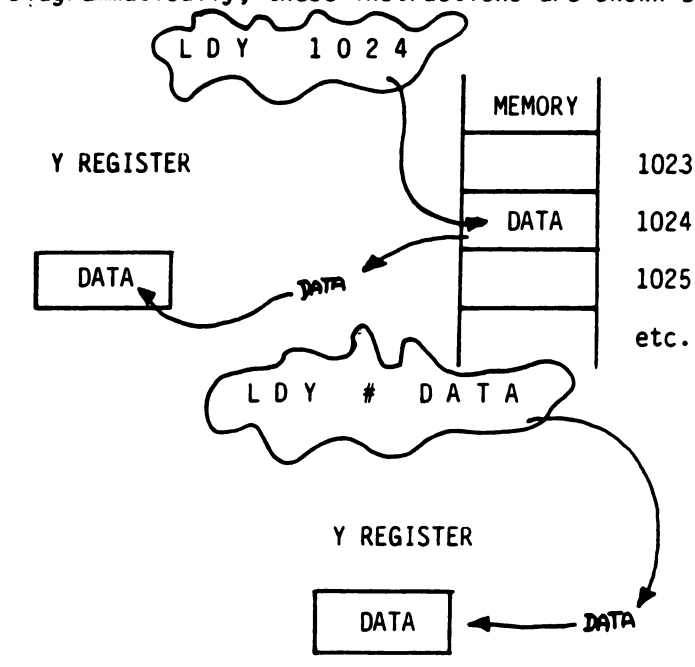
You will need to know that the screen of the APPLE //e normally occupies the 960 memory locations from 1024 to 2023. The 960 memory locations is needed to provided 24 rows each of 40 characters. As we have seen, 1024 is the top left hand position of the screen, and since 1026 was the third position ...

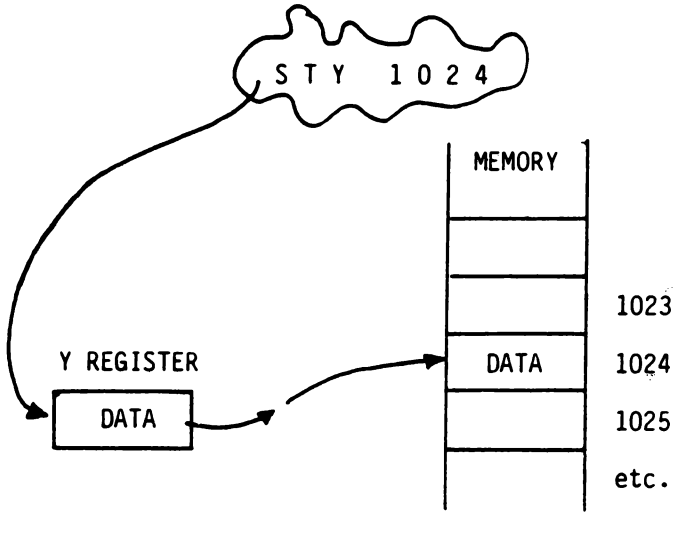
> EXERCISE 1.3
>
> Put an 'X' in each of the four corners of the screen. Answer in Chapter 9.

The Load and Store instructions that we have met so far are complemented by the corresponding Y-register instructions.

LDY	LoaD register Y with data at specified address.
LDY #	LoaD register Y with data specified in Immediate Mode.
STY	STore the data in the Y-register at specified address.

Diagrammatically, these instructions are shown below:-





You should now know, or be able to interpret the following:-

LDA	LDX	LDY	RTS
LDA #	LDX #	LDY #	
STA	STX	STY	
ADC			

For many operations, BUT NOT ALL, the X and Y registers can be treated interchangeably; for instance program 1.3 could be written:-

PROGRAM 1.3

```
START ADD? 768
LDA #1
STA 1024
LDX 1024
STX 1026
RTS
END
```

or

PROGRAM 1.3a

```
START ADD? 768
LDA #1
STA 1024
LDY 1024
STY 1026
RTS
END
```

Because of this interchangeability and the need to swap data rapidly between registers during the run of a program several instructions exist to do this automatically. They are typified by:-

TAX	Transfer the contents of the Accumulator into the index register X

Using this command in program 1.3 (to produce program 1.4), makes the program a little shorter but manages to achieve the same result.

PROGRAM 1.4

```
START ADD? 768
LDA #1
STA 1024
TAX
STX 1026
RTS
END
```

When this program is run the screen should display two white "A"s, one in 1024 and one in 1026.

Descriptions given of the codes so far have been spelled out in detail. However, as you are getting more used to the jargon, it is reasonable to begin to abbreviate. From now on, instead of "the contents of the X register", we will just refer to X and similarly so with the Y-register (Y) and accumulator (A). Thus, a summary of the transfer instructions is:-

TAX	Transfer A into X.		
		TAY	Transfer A into Y.
TXA	Transfer X into A.		
		TYA	Transfer Y into A.

EXERCISE 1.4

Write a program that loads a "Z" into the accumulator and an "A" into the X-register. Then, without using any further Immediate Mode commands, swaps these over and prints the "Z" on the first screen memory location and the "A" on the last.
A possible answer in Chapter 9.

EXERCISE 1.5

Write a program that: Loads an '!' into the accumulator, an asterisk into X and an "E" into Y. Then, without using any further Immediate Mode commands, moves the "E" into A, the '!' into X and the asterisk into Y. Print the '!' in the screen bottom-left, the asterisk in the bottom-right and two "E"s, one in each of the top two corners of the screen.

A possible answer in Chapter 9.

Hint: There are 40 characters in the APPLE //e screen line, so the top right hand corner of the screen memory is located at (1024 + 39). I leave it to you to work out the address of the bottom left hand corner.

***

CHAPTER 2

Jumping, Branching, Flags

FEW REAL LIFE PROGRAMS PROCEED ALONG A SMOOTH UNINTERRUPTED path without jumping or branching at some stage. This chapter looks at those commands and their uses and then examines the flags that enable the branches to be controlled.

Unconditional Jumps

These tell the program to jump willy-nilly - no conditions. Only two such 6502 instructions exist; the first to be considered is:-

JMP	JuMP to the specified address.

For instance, JMP 774 means jump to memory location 774.

Put in a program and it will look like this:-

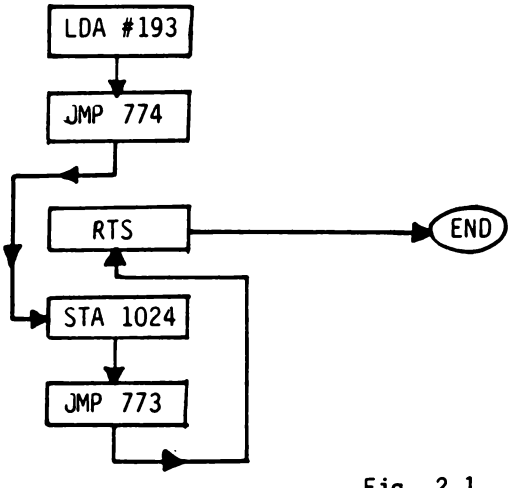


Fig. 2.1

Such a jump routine doesn't really achieve a lot but it could, for instance, be used to patch a piece of code into a program. In figure 2.1, for instance, the command STA 1024 has effectively been inserted into the program.

Now this can be typed in thus

PROGRAM 2.1

```
START ADDRESS? 768
LDA #193
JMP 774
RTS
STA 1024
JMP 773
END
```

Once again, it can be run by selecting 'R' on the menu and then starting the program at 768. When run, it should give an "A" in the top left hand corner of the screen.

When the jumps are used in this way it's necessary to tell the program exactly where to jump to, i.e. to give an ADDRESS, hence JMP 774. Calculating these addresses is quite straightforward as long as it is done systematically. For instance, all the commands or "SOURCE-CODES": RTS, LDA #, JMP, etc., take up one byte of memory, thus, to jump over RTS in Fig. 2.2a we jump from 770 to 774, i.e. over 773 which contains RTS.

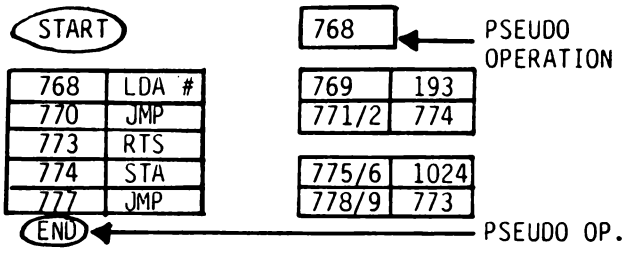


Fig. 2.2a

The part following the source-code is known as the OPERAND and calculating its length is a bit more complicated!

The easy way is simply to look it up in Appendix 1. Here you will find a complete list of all the source codes available on the 6502 microprocessor. For instance, at the bottom of page A1-19 you will find the entry for the RTS instruction. Under the heading NO. BYTES OPER. (number of bytes in operand) you will find the value zero. Thus, as we know already, RTS doesn't have an operand. It doesn't need one as the address to which it points is determined by the point from which the sub-routine came originally. On page A1-13 you will find an entry for LDA # which takes a single byte operand. The accumulator can only hold values up to 255, a single byte, in its eight bit register. There in the table for LDA #is a 1 to confirm this. Other instructions that you have met, require two bytes as their operands are greater then 255. For instance, JMP has an entry on page A1-12 showing that it has a two byte operand, and so on. By the way, I know the list of source codes is rather frightening, but DON'T PANIC, take them as they come.

Figure 2.2a shows the location of the various instructions and operands for Program 2.1.

LDA #193	takes up two bytes - one for its object code A9 and one for the number to be loaded into the accumulator. Remember the accumulator is only one byte long, so it can only hold a number up to 255.
JMP 774	takes three bytes - one for JMP (4C) and two for the address - here 774.
RTS	takes only one byte (60) - it has no operand as do TAX, TXA, etc.
STA 1024	This takes three bytes - one for STA (8D) and two for 1024
JMP 773	takes three bytes - one for JMP(4C) and two for the address - here 773.

This can also be seen by using the "LIST" command on the assembler MENU. Return to MENU and type "L" for LIST, then tell the assembler where to start listing, i.e. type in "768".

The screen will display:-

```
768 $0300   A9 C1       LDA #193
770 $0302   4C 06 03    JMP 774
773 $0305   60          RTS
774 $030    8D 00 04    STA 1024
777 $0309   4C 05 03    JMP 773
```

Fig. 2.2b

As Fig 2.2b shows, the assembler breaks the instructions and operands down into one-byte chunks. We can calculate the total length of the program by counting the one-byte pieces of the machine code. Thus, Program 2.1 is 12 bytes long:- A9 C1 4C 06 03 60 8D 00 04 4C 05 03

JSR	Jump to Sub-Routine.

This is another jump command which is used along with RTS and together these are like GOSUB.....RETURN in BASIC.

```
        BASIC                ASSEMBLER

10      GOSUB 200              768  JSR 774
          "
          "
          "
          "
200     REM***SUB-ROUTINE      774  STA 1024
          .    .    .          .    .    .
300     RETURN                 777  RTS
```

We can modify Program 2.1 to use this instruction instead of the straight JMP used there. The program then becomes:-

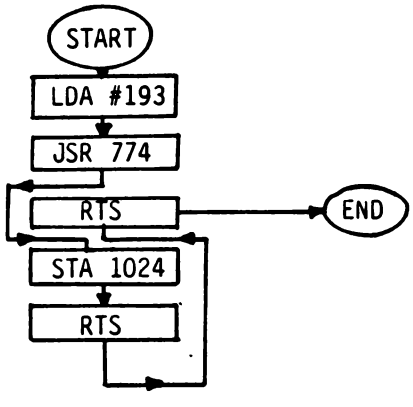


Fig. 2.3

PROGRAM 2.2

START ADDRESS? 768	Pseudo-code
LDA #193	Load Accumulator in immediate mode with '193'.
JSR 774	Jump to the subroutine at 774.
RTS	Return from subroutine (i.e. back to BASIC).
STA 1024	Store contents of accumulator in 1024.
RTS	Return from subroutine.
END	Pseudo-code

The advantage of RTS over JMP is shown in this program, as with RTS it is not necessary to calculate the address for the jump which organises the return to the main line of the program. In Program 2.1 we had to put in the JMP 774 to have the same effect as RTS in this program. The 6502 does this trick by use of the

PROGRAM COUNTER (PC)

This is a 16-bit register which contains the address of the next command which is to be executed. In reality all it is is two 8-bit memories, one for each byte, which is built into the 6502 chip. When you select 'R' at the assembler MENU and then type in 768, this generates a command that sets PC to 768 and starts execution from there. As the PC fetches each byte from memory it is incremented by 1, thus always pointing to the next memory location containing the required data.

Take the first three lines of Program 2.2 for instance:

```
START 768
LDA #193
JSR 774
```

A summary of the PC counter contents during execution of this is given in Fig. 2.3.

PROGRAM	PC BEFORE EXECUTION	PC AFTER EXECUTION
START 768	?	768
LDA #193	768	770
JSR 774	770	774

Fig. 2.3

This figure illustrates how the PC steps through the program until it comes to the JSR command. It then takes the jump command and sets the PC to the address specified, i.e. 774. As it is only two bytes long and can thus only store one address, the PC uses some external memory when it needs to remember more than one. This area of memory, the stack, is discussed below (page 7-7).

> EXERCISE 2.1
>
> Write a program to put a 3 in the accumulator. The program is to start at 768, then jump to a sub-routine at 900 which adds 3 to the 3 already in A, return to original routine and print the accumulator sum onto the top left-hand corner of the screen. *
>
> Answer in Chapter 9.

Conditional Jumps

We have already looked at unconditional jumps but any program that needs to test for conditions needs CONDITIONAL JUMPS. In BASIC, the analogy is with the IF...THEN command,

i.e. 10 IF X=Y THEN 500

In this line the values X and Y, which have been stored in memory, are compared.

The 6502 carries out this operation in several different ways - one of these is by using a special register known as the STATUS REGISTER (SR), sometimes known as the PROCESSOR STATUS WORD. The SR is an eight bit register like the accumulator and X and Y register, but it is used quite differently from these. Whereas the other registers are used to store and manipulate bytes, the SR is treated as if it contained eight individual bits which are used as signals or flags. The 6502 normally only handles one status flag (as they are known) at a time, either setting the bit value to '0' or '1', or testing the status flag to determine whether it is set ('1') or cleared ('0').

One example of the status flags is the Z flag or the ZERO flag. Whenever an arithmetic process (or just a move) is carried out that produces a result of zero in the appropriate register (A, X or Y) then the Z flag is set to '1'. If, on the other hand the result of the process is non-zero then the Z flag is set to '0'. Look at it this way: if the flag is SET (to '1') then the condition (or status) is TRUE, if the flag is CLEAR (to '0') then the status is FALSE. The Z flag is concerned witn a zero condition or status, so Z flag set means that it is true that the condition is zero.

Several different instructions can set this flag, one of these being:-

DEX	DEcrement the contents of the X register.

Tne segment of Program 2.3 below demonstrates this in use.

PROGRAM 2.3
(In Part)

```
START ADDRESS? 768
LDX #100
DEX
```

It loads X with '100' and then decrements this one down to 99. This facility for indexing both the X and Y registers accounts for their name - index registers. When the contents of X are zero, the zero flag is set to 1. If we wish to use the setting of this flag to control the program then we must use an instruction that tests the flag and brings about a branch dependent upon whether or not it is set. Such an instruction is:

BEQ	Branch if result was EQual to zero. (i.e. if the ZERO flag is set.)

Tnis checks on the status of the Z flag and if it is set (to 1), branches as specified. The operand in this case in only one byte long, so only 0 to 255 can be accommodated. As these 256 numbers are needed for branching in both directions 0 to 127 are assigned to forward jumps, for example '60' giving a forward jump of 60 steps while 128 to 256 are used for backward steps. In the case of the latter a branch instruction of, say, 200 gives a backward step of 256 - 200 or 56 steps.

The Dr Watson Apple assembler kindly lets you forget these rules about forward and backward steps. It works out the branch steps for you and provided the branch is within 127 bytes, the computer is happy. So, BEQ can be used as in Program 2.3 ie. BEQ 776, where this means branch to 776 (if equal). BEQ 776 is carefully replaced by BEQ 3 on assembling this program.

This means "jump forward 3 places or bytes". Don't forget when using branches that a branch command is restricted, while those of JMP and JSR are not.

The BEQ instruction is illustrated in Program 2.3 below, where it checks the condition of the Z flag, and, if set, branches forward 3 bytes.

PROGRAM 2.3

```
START ADDRESS? 768
LDX #100
DEX
BEQ 776
JMP 770
STX 1024
RTS
END
```

When run this program prints an inverse "@" in 1024.

As with many X-register instructions, DEX has a corresponding Y register instruction:-

DEY	DEcrement the contents of the Y register.

> EXERCISE 2.2
>
> Write a program to carry out the same operation as Program 2.2 but utilising the Y register. Answer in Chapter 9.

A second instruction that also checks the Z flag is:

BNE	Branch if Not Equal.

This does the reverse of the BEQ command and branches if the Z flag is NOT set. Program 2.4 below, is a modification of Program 2.3. Notice how the original program is shortened considerably by the use of BNE rather than BEQ.

PROGRAM 2.4

```
START ADDRESS? 768
LDX #100
DEX
BNE 770
STX 1024
RTS
END
```

When run, the program is identical in effect to Program 2.3 and puts an inverse "@" in 1024.

The index registers have been indexed downwards by the DEX and DEY commands. As you might expect, they can also be indexed upwards. This is done by means of:-

INX.	INcrement the contents of X by one.
INY	INcrement the contents of Y by one.

Instructions to Compare Values

Naturally, when incrementing, a straight check for zero is not possible so the registers must be compared against a value previously set somewhere and the 6502 possesses three instructions to do this. The first of these instructions to be examined is:-

CPX	ComPare the contents of the specified memory address with the X register.

This is actually done by subtracting the memory contents from X and can thus give a positive, negative or zero value. Thus, the instruction CPX 890 does the following:-

1. Read contents of memory location 890.
2. Subtract these contents from those of the X register.
3. Set Z flag if answer=0. (Also sets other flags not yet considered.)

NOTE:

Neither the contents of the memory location nor the X register are changed during this operation.

At the moment we are interested in the zero condition. To utilise this instruction we can set the X register at zero and store a value for comparison in memory somewhere. The flow diagram for this is given below, in Fig. 2.5.

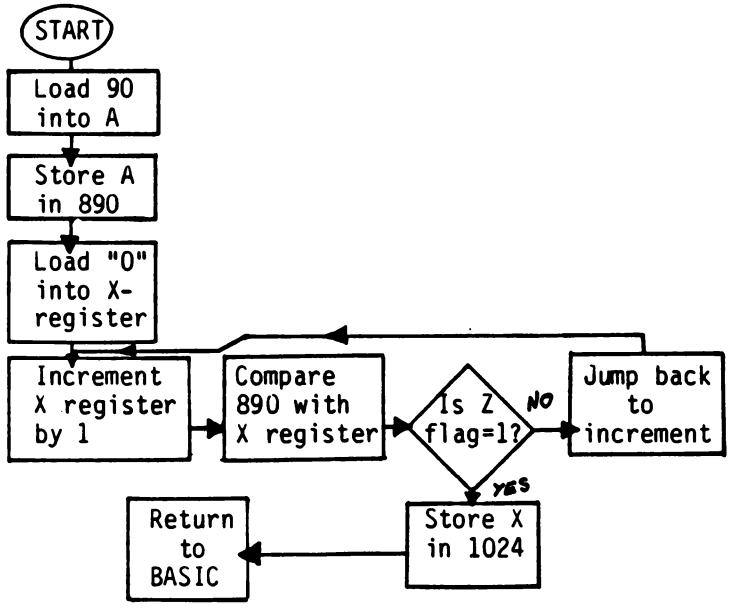


Fig. 2.5

Written into a program it looks like this:

PROGRAM 2.5

```
START ADDRESS? 768
LDA #170          Load 170 (an asterisk) into
                  Accumulator.
STA 890           Store contents of Accumulator
                  in 890.
LDX #0            Load '0' into X register.
INX               Increment X register.
CPX 890           Compare value in X register
                  with that in 890 (i.e. 90).
BEQ 784           Branch forward three bytes if
                  CPX answer=0.
JMP 775           Jump to memory location 775.
STX 1024          Store contents of X in 1024.
RTS               Return from machine-code to
                  BASIC.

END
```

Just to make that clearer, we'll step through some of the stages. Figure 2.6 below is numbered in stages, where stage 1 represents the first time that the program steps through INX, stage 2 the second time, and so on.

Note that although X is loaded with a '0', this is immediately incremented to a '1' at INX.

Loop Number	Accumulator contents	X-register contents	Z-flag
1	90	1	0
2	90	2	0
3	90	3	0
etc.			
88	90	88	0
89	90	89	0
90	90	90	1

Fig. 2.6

At stage 90, BEQ is activated and program jumps 3 bytes to STX 1024 and then RTS.

Now type in the program and run it. It should display an asterisk in 1024.

The compare instruction CPX has a corresponding instruction for the Y register:

CPY	Compare the contents of specified location with those in the Y register.

Its operation corresponds exactly with that for CPX.

EXERCISE 2.3

Re-write Program 2.5 to use the Y-register rather than the X and on completion of the loop print out a question mark at 1034.

Answer in Chapter 9.

The third compare instruction is:

CMP	CoMPare the contents of the specified memory with the Accumulator.

This is particularly useful as the results of all arithmetic operations are deposited in the Accumulator and CMP allows a direct comparison between a specified value and an 'answer'.

An example of its use is given in Program 2.6.

PROGRAM 2.6

```
START ADDRESS? 768
LDX #0              Load a '0' into X.
LDA #191            Load a  '191' (a question mark)
                    into A.
INX                 Increment X.
STX 900             Store X in 900.
CMP 900             Compare A with 900.
BNE 772             Branch if Not Equal.
STX 1024            Store X in 1024.
RTS                 Return from Subroutine.
END
```

Let us have another look at Program 2.5. It turns out that using CPX to put a '1' into the Z flag was a bit like using a sledgehammer to crack a rather small nut, as the Z flag is very readily set. In fact, the Z flag is set to '1' whenever a zero is passed from memory to the Accumulator, to the X or to the Y registers or when a zero answer is obtained to an arithmetic process. Thus, the Program 2.5 could have been written omitting the CPX 890 instruction.

To demonstrate this, Program 2.5 is re-written using the Y register and DEY and testing with the command BNE, as shown in figure 2.7 below:-

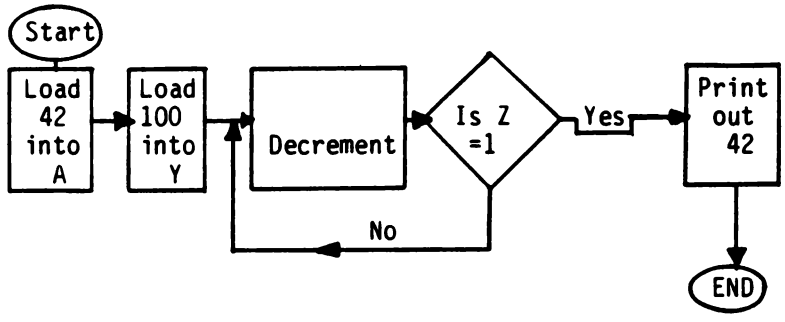


Fig. 2.7

And the program:-

PROGRAM 2.7

```
START ADDRESS? 768
LDA #170
LDY #100
DEY
BNE 772
STA 1024
RTS
END
```

The instruction BNE 772 gives a backward branch of 3 taking the program back to DEY, as the step is counted from the beginning of the next instruction whether the branch is forward or backward.

If you haven't run Program 2.7 by now, have a go, it should put an asterisk in 1024.

EXERCISE 2.4

Now write the program for the flow diagram below, figure 2.8:-

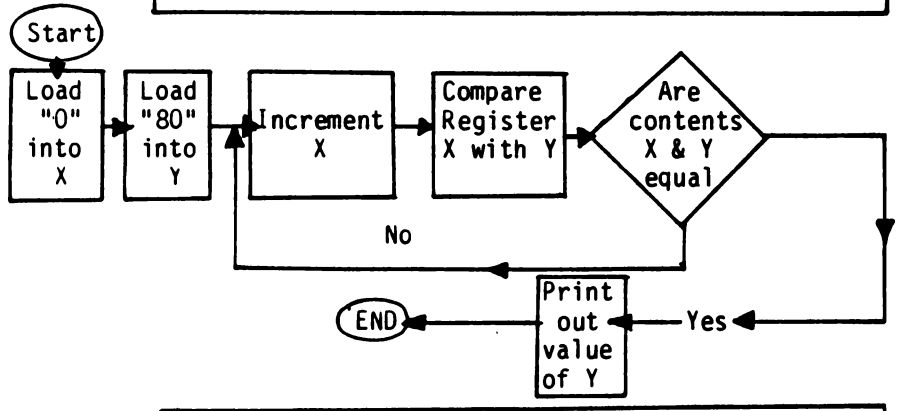


Fig. 2.8

If you run into difficulty with this, the more detailed flow diagram in Chapter 9 should help. PLEASE don't look at this until you've had a try, though.

The answer is given in Chapter 9.

The 6502 Flags

Program 2.7 used the Z-flag which is only one of 7 flags available on the 6502. As these each only contain one BIT of data, i.e. a '0' or a '1', they can all be stored in one byte of memory - the Processor Status Register. Thus the flags are contained in the SR as shown below in Figure 2.8.

Bit number	7	6	5	4	3	2	1	0
Flag	N	V	-	B	D	I	Z	C

Fig. 2.8

It is not proposed to give a full description of all these now; they will, however, be described and illustrated when they are encountered. The Z-flag - the Zero flag - has, of course, already been met and used in programs and exercises.

The function of the flags is summarised below:-

N — Negative flag. Set when an arithmetic operation results in a negative result. The flag is controlled by the instruction ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TXA, TYA.

V — Overflow flag. Set when an arithmetic operation results in an overflow from bit 6 to bit 7, tells that result will be wrong unless overflow allowed for. The flag is controlled by the instruction ADC, BIT, CLV, PLP, RTI, SBC.

B — Break flag. Set when a programmed interrupt is brought about by a BRK instruction.

D — Decimal flag. Set when arithmetic operations are to be carried out in decimal. The flag is controlled by the instructions CLD, PLP, RTI, SED.

I — Interrupt flag. Set when an interrupt sequence is in operation. The flag is controlled by the instructions BRK, CLI, PLP, RTI, SEI.

Z — Zero flag. Set when an arithmetic operation results in a zero answer. The flag is controlled by the instructions ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TXA, TYA.

C The Carry flag. Indicates the presence of a 'carry' or a 'borrow' during arithmetic operations. Also set during shift or rotate operations to indicate possible loss of a bit. The flag is controlled by ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

The N-Flag

The N-flag, bit 7 on Fig. 2.8 above, is the Negative flag which i set to '1' when the answer to an operation is negative. It can b tested by two instructions, one of which is:-

BMI	Branch on MIinus.

An instruction such as BMI would test the N flag and if set branc the program backwards, or forwards, as specified.

An example of the use of BMI is given in Program 2.8, where th contents of Y are incremented until a BMI check gives a branch t the STY 1024 which displays the current value of the Y-register o the screen. This display is an @ sign which corresponds to a 128 Thus the program increments the Y value step-by-step up to 127 an the computer recognises all these numbers as positive. However, a it steps from 127 to 128, the minus flag is set as numbers from 12 to 255 are recognised as negative - just the same as with branches ie. 1 to 127 forward, 128 to 255 back. This happens because th numbers are stored in what's known as two's complement - but mor of that in Chapter 8.

PROGRAM 2.8

```
START ADDRESS? 768
LDY #0
INY
BMI 776
JMP 770
STY 1024
RTS
```

When run, this program will put an @ sign (a 128) in 1024.

A similar sort of effect, but working the other way is obtained b means of:

BPL	Branch on PLus.

This instruction tests the N flag and if it is NOT set, causes the program to branch. In Program 2.9, the Y-register is loaded with a 255 and this sets the negative flag. This register is then decremented progressively and, as it steps from 128 to 127, the N flag is cleared and the program branches on the BPL instruction.

PROGRAM 2.9

```
START ADDRESS? 768
LDY #255
DEY
BPL 776
JMP 770
STY 1024
RTS
```

When run, this program puts a flashing question mark (a 127) in 1024.

CHAPTER 3

More Instructions, Addressing, Screen Outputting

ONE OF THE ADVANTAGES OF MACHINE-CODE PROGRAMS IS THEIR speed of operation and this naturally facilitates screen displays. Animation, for instance, can be achieved by means of commands such as:-

STA ...,X	STore the contents of the Accumulator in the specified address indexed with the X register.

(...means an address goes in here, read on)

Thus, if X contains 100 and the accumulator 218, the instruction

```
STA 1024,X
```

will put a 'Z' into (1024+100). When used along with the increment instruction, this enables the location on the screen to be indexed. Program 3.1 demonstrates this.

PROGRAM 3.1

```
START ADDRESS? 768
LDX #100              Load 100 into X.
LDA #218              Load 218 (Z) into A
STA 1024,X            Output Z at (1024+X)
DEX                   Decrement X value
BNE 772               Branch if not equal
RTS
END
```

When run, this program puts a 'Z' in the first 100 locations in screen memory, which is not the first 100 locations on the screen! Once again, the 'X' command has a corresponding 'Y' command:

STA ...,Y	STore the contents of the Accumulator in the specified address indexed with the Y register.

EXERCISE 3.1

Modify program 3.1 to use the Y register rather than the X register, using only direct POKE commands.
Answer in Chapter 9.

Instead of the decrement instruction being used, the program could have used an increment but would then have had to do a compare with 100 in order to set the zero flag to a '1'. Try this as an exercise!

EXERCISE 3.2

Print an asterisk in the first 100 screen locations using an INX command to increment.
A possible answer in Chapter 9.

In Exercise 3.2 the branch instruction was activated by zero generated by a compare command. However, if the X or Y register is incremented from 255, it clocks back to zero and resets the Z flag. It can thus be used to branch without a compare if it is initially set to the appropriate value. Program 3.2 performs a similar function to 3.1 but uses INX rather than DEX. It increments X from 216 to 255 and on the 39 loops through, plus the next loop, prints out an asterisk. Once at 255 the 8 bit register is full of '1's and the addition of one more '1' ripples through each bit of the store, resetting them all to '0's. The 6502 does notice this switching over and sets a flag to remember the event (discussed above, page 2-14).

This program offers no advantage over 3.1 on its own but may in a particular context be advantageous.

PROGRAM 3.2

```
START ADDRESS? 768
LDX #216
LDA #170
STA 1024,X
INX
BNE 772
RTS
END
```

As with BASIC programs a character can be moved across the screen by filling the screen with the character while POKING a blank one space behind it. Program 3.3 demonstrates this type of routine.

PROGRAM 3.3

```
START ADDRESS? 768
LDX #0
LDY #160
STY 900
LDA #32
STA 901
STA 1024,X
TYA
STA 1023,X
LDA 901
INX
BNE 780
RTS
END
```

When run, this program runs a white square across the screen to 1279. Unfortunately location 1279 is one of the 40 screen locations NOT displayed so it is not easy to know if the program has worked!

You may be quite at home with this general technique of animation; if so please ignore figure 3.1. If not, you should step through this stage by stage.

Command	Acc	X-Reg	Y-Reg	900	901	Screen mem.	Content	Z-flag
LDX #0	?	0	?	?	?	?	?	0
LDY #160	?	0	160	?	?	?	?	0
STY 900	?	0	160	160	?	?	?	0
LDA #32	32	0	160	160	?	?	?	0
STA 901	32	0	160	160	32	?	?	0
STA 1024,X	32	0	160	160	32	1024	32	0
TYA	160	0	160	160	32	1024	32	0
STA 1023,X	160	0	160	160	32	1023	160	0
LDA 901	32	0	160	160	32	1023	160	0
INX	32	1	160	160	32	1023	160	0
BNE 780	32	1	160	160	32	1023	160	0
STA 1024,X	32	1	160	160	32	1025	32	0
TYA	160	1	160	160	32	1025	32	0
STA 1023,X	160	1	160	160	32	1024	160	0
etc.	until...							
STA 1024,X	32	255	160	160	32	1279	32	0
TYA	160	255	160	160	32	1279	32	0
STA 1023,X	160	255	160	160	32	1278	160	0
LDA 901	32	255	160	160	32	1278	160	0
INX	32	0	160	160	32	1278	160	1
BNE 780	32	0	160	160	32	1278	160	1
RTS	32	0	160	160	32	1278	160	1

Fig. 3.1

As written, Program 3.3. is by no means the only way of doing the job and as you no doubt observed, it is a long way from being the best; nevertheless it does flash the white square to where it's required. Later on in this chapter we will develop a more acceptable version of the program, but in order to do this we must first look at the problems associated with...

The Timing of Programs

Program 3.3 highlights one of the problems of machine code - a few pages ago it was an advantage! - SPEED. Whereas in BASIC it's not often necessary to slow things down, that's not so in machine code. The 6502 chip takes its operating speed from an internal clock driven from a crystal oscillator, which in the case of the APPLE runs at 1 MHz (one MegaHertz) or one million cycles per second. Thus each cycle takes 1 millionth of a second and speeds of operation of the various instructions will be referred to by the number of cycles that it takes for them to be carried out, or EXECUTED. Some of these operations take place entirely within the 6502 and are carried out much quicker than those which are required to retrieve data from memory. The instruction TAX, for instance, takes two cycles to execute while STA ...,X takes five.

Clearly, a knowledge of the time taken for the instructions to be executed is important as it is this that determines the speed of operation of the program and also allows use to be made of the 1MHz clock for timing loops and delays.

Looking back at Program 3.3, the time that a character remains on the screen prior to "moving on" can be calculated. Thus, the white square appears at STA 1024,X and the next few stages are tabulated below:-

Command	Execution time (cycles)	Elapsed Time (cycles)
STA 1024,X	-	0
TYA	2	2
STA 1023,X	5	7
LDA 901	4	11
INX	2	13
BNE 780	3	16
STA 1024,X	5	21
TYA	2	23
STA 1023,X	5	28

So, from a white square appearing to its being overwritten by a blank is 28 cycles or 28 micro-seconds. Overall, the 256 white squares are written in about 7168 micro-seconds or 7.2 milli-seconds, much faster than the eye can follow. Indeed, since the television screen is only scanned once every 20 milli-seconds (European PAL system) or 16.7 milli-seconds (USA NTSC system) then this is much faster than your television screen can follow.

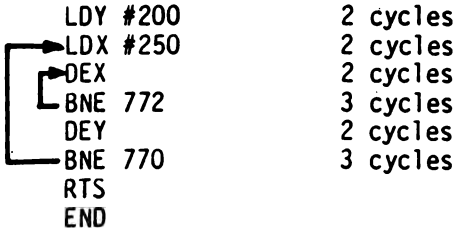
To attain a more leisurely progress across the screen a delay could be programmed in to allow the white square to stay in view longer. Such a device is simple in principle but may need some care when implementing in actual application. Program 3.4 below shows a simple delay loop.

PROGRAM 3.4

```
START ADDRESS? 768
LDX #250          2 cycles
DEX               2 cycles
BNE 770           3 cycles
RTS
END
```

This gives (ignoring LDX #'s 2 cycles) a delay of 5 cycles per loop, or 250x5=1250 cycles per execution. Even when run right through, a delay of only 250 micro-seconds is obtained and this must be augmented by nesting this loop within another one as Program 3.5 shows.

PROGRAM 3.5



A complete run of this program would run the DEX subroutine 200 times, i.e. achieving an overall delay of 200 x 1250 micro-seconds, or ¼ of a second.

If we wish to use the computer for precision timing, then we can clearly not ignore the odd two micro-seconds here and there and we must step very carefully through the program to ensure that we account for all parts of the program. In particular we must watch the branch instructions. For instance, the BNE in program 3.5 normally takes three cycles, i.e. when the branch succeeds. However, when the branch fails and the program runs past the instruction it takes only two cycles. Under other conditions, if the branch takes the program into another section of memory (i.e. another "page", see page 3-9 for discussion), the instruction takes an extra two cycles!

However, we were principally interested in delays in order to slow down our animation so let's try putting some delays into a program (3.3) to check that they really do work!

Program 3.6 uses 3.3 as a basis and inserts the delay loop illustrated in 3.4 putting 1.2 milliseconds between the appearance and disappearance of a white square.

PROGRAM 3.6

```
START ADDRESS? 768
LDY #0
LDA #32              Set up a white square
STA 900
LDA #160             Set up a blank
STA 901
LDX #250             Set up for delay
LDA 900              Load and display
STA 1024,Y           white square
DEX                  Delay
BNE 787              Loop
LDA 901              Load blank into accumulator
STA 1024,Y           Display blank
INY                  Set up to process next
BNE 780              location on the screen
RTS
END.
```

Try entering this program and running it. Disappointing isn't it? The truth of the matter is that 1.2 milli-second simply isn't long enough. The television screen is only refreshed every 1/50th second (European PAL) or 1/60th second (USA NTSC) so it takes roughly 16-20 milliseconds for the screen to be scanned. If our little white square is only on the screen for approximately one thirtieth of that time, the chances are that only one in thirty of our diamonds is going to be seen (i.e. one or two a line). If you look carefully, and you have to look carefully, you will detect that the situation is a bit better than that, roughly three or four a line. My engineering friends tell me that this is because of something called 'interlace' and I have to believe them. In any case, it appears that the delay produced by Program 3.6 is therefore about fifteen times too short.

How then can we increase the delay? The X register can only hold a maximum of 255 and we are already using a count of 250, so there isn't much scope for increasing the number of times around the loop unless we use the double loop illustrated in Program 3.5. But the extent of the delay depends on the time used by the loop as well as the number of times round the loop. The DEX/BNE loop uses 5 cycles for each loop, could we make it use more? The answer is yes. If the BNE branch at the end of the delay loop went back to the LDA 900 instruction it would increase the number of cycles consumed by the loop to 23. This would mean that each of the white square would appear on the screen for 3 milli-seconds, still not enough time for the 16-20 milli-second scan to see every white square. Try modifying the program so that the BNE of the delay loop goes back to the LDA 900. When you run it again you should see roughly one in three of the white squares.

We need to increase the delay by a factor of fifteen in the original program (or a factor of three in the modified version). One possible solution is to use the double loop of Program 3.5. The problem is that Program 3.6 uses the Y register for indexing the character across the screen and we cannot therefore use the Y register for the outer loop of the double loop, or can we? Well we can, of course. It is always possible to save the value in the Y register in memory prior to entering the delay loop and then retrieve it after the loop. When used in this way, the lack of more than two registers in the 6502 can be overcome quite readily - at the expense of a little more coding.

PROGRAM 3.6A

```
START ADDRESS? 768
LDY #0
LDA #32              Set up a white square
STA 900
LDA #160             Set up a space
STA 901
LDA 900              Load up a white square
STA 1024,Y           Display it on the screen
STY 902              Save Y register during...
LDY #15              Set outer loop
LDX #250             Set inner loop
DEX
BNE 793              Count down 250 times
DEY
BNE 791              Count down 15 times
LDY 902              Restore Y register as screen index
LDA 901              Load blank
STA 1024,Y           Display blank
INY                  Set up for next screen location
BNE 780              Loop unless all done
RTS
END
```

Enter this program and run it. Brilliant isn't it! It really demonstrates why computer games which are written in machine code are so much better than those written in BASIC. Bear in mind that we slowed the machine code down by adding 4000 delay cycles on to each 27 useful work cycles and you can imagine how much manipulation could be carried out using machine code.

Modes of Addressing

When moving a character across the screen we have used the instructions STA ...,X and STA ...,Y as these are able to index along with the relevant register. In earlier work we used the STA instruction on its own and this clearly a relative of STA ...,X and STA ...,Y. The difference between the two types of instruction lies in their MODES of addressing and clearly the X and Y are part of this. In fact, the STA instructions has seven varieties dependant on the mode of addressing used. Thus, the addressing mode is a modifier of the command, designed to modify its function in a particular way.

The address essentially points the 6502 to a location in memory either directly or indirectly, the way it does this being determined by the particular mode of addressing used. This process is uniform throughout the 64K of possible memory except for the 256 locations from 0 to 255. To address these locations only one byte is needed whereas all other locations need two. Hence this area is given a special name - ZERO PAGE - and a special mode of addressing. In fact the whole of memory is divided up into 'pages' of 256 bytes and an instruction that causes operation over the boundary between pages takes an extra cycle to be executed.

Using the Dr.Watson assembler it is not normally necessary to worry about zero page addressing as the assembler will use it automatically when you address a memory location that is in zero page, (i.e. between 0 and 255).

Instructions	taken as zero page?
STA 900	NO
STA 43	YES
LDA 129	YES
LDA 1024	NO

This does NOT mean that you can forget zero page completely - it has other special uses!

Implied Addressing

This mode, sometimes also known as inherent addressing, is probably the easiest to use, as the 6502 does all the work for you!

Several instructions have already been used as TYA, TXA, RTS, in which the 6502 itself calculates the address. Basically they form two separate groups, one in which the whole instruction is executed within the 6502 itself, i.e. TYA, transfer Y to A and the other group where an external reference is necessary, e.g. RTS return from subroutine.

The members of the first group already considered are: DEX, DEY, INX, INY, TAX, TAY, TXA, TYA, while those yet to be discussed are: CLC, CLD, CLI, CLV, NOP, SEC, SED, SEI. Those in the second group are RTS (already used) and BRK, PHA, PHP, PLA, PLP and RTI.

Absolute Addressing

Instructions using this mode are among the easiest to understand as their operand is a two byte number that defines the address absolutely. Thus in program 3.6, page 3.7, the instruction STA 901 tells the A register exactly where to store its contents. Later in the same program LDA follows the same pattern.

Instructions that utilise this form of addressing are: those already met; ADC, CMP, CPX, CPY, JMP, JSR, LDA, LDX, LDY, STA, STX, STY, while those yet to be discussed are: AND, EOR, ORA and SBC.

Zero-Page Addressing

This form of addressing is really a sub-set of absolute addressing but is restricted in size of its operand to 255. The major advantage of this mode is that it executes in only three cycles compared with the four required for the absolute modes. Because of the faster execution times when using page zero addresses, page zero of the APPLE is used by the BASIC interpreter and is, thus, not readily available for machine code programs. A few of these zero page locations are not used by BASIC and may be used by the machine code programmer, notably 251 to 254. When you get to know your way around the BASIC interpreter then you will discover that you can use a lot more. Finally, it is possible to use the zero page by relocating this page to another location in RAM but this is felt to be a procedure beyond the scope of this book.

Immediate Addressing

This mode of addressing allows a number to be loaded immediately, i.e. directly as specified, into a register or to be used directly as a means of comparison. All the immediate commands in this book are recognisable by the '#' in their source code.

Many examples of immediate mode addresses have been seen already, for example in Program 3.6, page 3.7. In this program, the Accumulator was loaded directly using LDA #32, and in other programs both the X register and the Y register have been loaded in the same way. However, many other instructions can be used in the immediate mode, resulting in neater programs. Program 3.7 demonstrates the use of:-

CPY # ComPare Y with value specified in immediate mode.

PROGRAM 3.7

```
START ADDRESS? 768
LDY #0
TYA
INY
STA 1023,Y
CPY #100
BNE 770
RTS
END
```

On running, the program prints the first 100 characters of the character set in the first 100 screen locations.

Indexed Addressing

In this mode, an address is calculated using the contents of a register added to a specified address. It has been used frequently to print characters across the screen in the form

'STA address,X','STA address,Y'.

In Program 3.7, STA ...,Y (...is short for address) was used in this way with the instruction STA 1023,Y.

Both registers can be used with absolute indexed instructions, i.e. operands with two bytes.

Those indexed with 'X'		Those indexed with 'Y'	
ADC ...,X	LDY ...,X	ADC ...,Y	LDX ...,Y
AND ...,X	LSR ...,X	AND ...,Y	ORA ...,Y
ASL ...,X	ORA ...,X	CMP ...,Y	SBC ...,Y
CMP ...,X	ROL ...,X	EOR ...,Y	STA ...,Y
DEC ...,X	ROR ...,X	LDA ...,Y	
EOR ...,X	SBC ...,X		
INC ...,X	STA ...,X		
LDA ...,X		... means a 16 bit address	

N.B. STY CANNOT be indexed with X (except in zero page).
STX cannot be indexed AT ALL. i.e. STY...,X and STX ...,Y DO NOT EXIST.

ASL, DEC, LSR, ROL, ROR CANNOT be indexed with Y.
i.e. ASL ...,Y and DEC ...,Y etc. DO NOT EXIST.

It is, of course, NOT possible to use the two X register index commands STX and LDX with reference to X itself.

Relative Addressing

Branch instructions generally use relative addressing, although we are protected from the full "horrors" of relative addressing by the assembler, which allows us to handle branches by using absolute addressing (see the introduction of branches 'BEQ' in Chapter 2).

It is the branch group of instruction that uses relative addressing and this group consists of BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

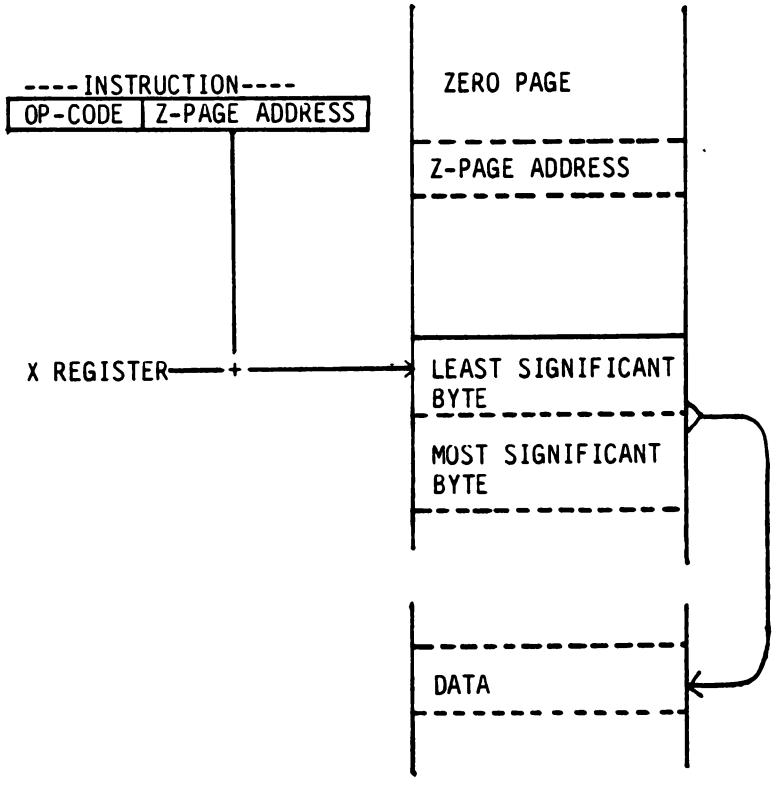
Indirect Addressing

This is by far the most complicated - and the most versatile of all the addressing modes. It gets the "indirect" in the name from the fact that the operand is a pointer and not an address. It is this pointer that directs the 6502 towards the memory location that contains the address.

However, once again, the X and Y indexing mechanisms differ considerably in operation and give rise to different sub-modes of addressing. All the instructions that utilise this kind of addressing are recognisable in the assembler as they contain either an (..,X) or a (..,Y) and have a one byte operand. Because of this they can only point to locations in the zero-page and hence suffer from the same restrictions as the other zero-page commands.

Using the X-Register

With indirect addressing using the X register, the contents of the X register are added to the second byte of the instruction to produce an address in zero page. The address is used point to a second set of two bytes that form the address of the memory location to be used by the instruction (to store data in or to take data from, etc.). This is illustrated in the diagram below.



This instruction is useful for cases when the programmer wishes to access a number of 'randomly' placed locations in memory. If a list of relevant locations' addresses is placed in zero page, it can be clocked through by the X register (by incrementing in steps of two, as it takes two bytes to make one address). Thus it can point to each of these 'randomly' placed locations in turn.

This type of addressing is known as Indexed Indirect Addressing, or perhaps more clearly as PRE-INDEXED INDIRECT ADDRESSING. As the latter name implies, the addressing is pre-indexed as the X value is added on before the 6502 picks up the address.

Instructions that use this type of addressing are:- ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

Using the Y-Register

Indirect addressing using the Y register operates somewhat differently, as the operand instruction points directly to a zero-page memory location. This contains the least significant byte of the address, the next memory location containing the most significant byte. Finally, the index register contents are added to this address to form the final indexed address. Not surprisingly, this form of addressing is referred to as POST-INDEXED INDIRECT ADDRESSING as the indexing is carried out after the address is retrieved. The BASIC interpreter and the APPLE operating system make extensive use of this instruction. When you have become an experienced Assembler user you will no doubt wish to look at the way the BASIC interpreter works and you will see how useful these instructions are.

Commands that use this type of addressing are:- ADC, CMP, EOR, LDA, ORA, SBC, STA.

Indirect Absolute Addressing

This mode of addressing is used by one instruction only:

JMP (...) JuMP indirectly addressed.

It is an absolute instruction in that the operand is a two byte address and can thus address any location in memory. However, it is indirect in that, at that location and the subsequent one it finds the address (LSB first then MSB) for the jump instruction.

Putting this into a program using JMP (784) yields:

PROGRAM 3.8 (Part)

```
DEC     HEX
780     $030C     JMP (784)
783     $030F     NOP
784     $0310     ORA 3,X
786     $0312     NOP
787     $0313     NOP
788     $0314     NOP
789     $0315     STA 1024
792     $0318     RTS
```

This program, on meeting JMP (784) jumps to 744 to pick up $15 and then to 785 to retrieve $03 and re-assembles the address $0315 (789). This is loaded into the program counter and the program jumps to 789 to execute the program stored there. As Program 3.8 (Part) stands it is not possible to program in the $15 and S03 into 784 and 785 as these hex numbers do not represent valid op-codes. The assembler will simply reject them. They can be most easily put in by loading them into a register and then transferring into memory - as is shown in Program 3.8 (whole).

PROGRAM 3.8 (whole)

```
START ADDRESS?  768
LDA #170
LDX #21
STX 784
LDX #3
STX 785
JMP (784)
NOP
NOP
NOP
NOP
NOP
NOP
STA 1024
RTS
```

When run, this first loads the address $0315 into locations 784 and 785. The JMP then retrieves these and jumps to $0315 where it executes the routine stored there. This, however, is a routine and not a subroutine as it was entered with a JMP and not a JSR, so the RTS at 792 returns the program to the BASIC program. Once run, this program will have modified itself and put the $03 and $15 into 785 and 784.

It is not considered good programming practice to write programs that modify themselves like Program 3.8 which is purely a demonstration of the JMP instruction.

Addressing Generally

The whole subject of addressing is clearly a complex one and one to be approached only with care. A basic rule must be to check carefully in Appendix 1 before using any addressing of which you are not absolutely certain. To some extent, the assembler will assist in weeding out instructions that don't exist but it can't write your programs!

CHAPTER 4

Mathematical, Logical Operators

ON PAGE 1-1, AS EARLY AS PROGRAM 1.1, A PROGRAM WAS WRITTEN to add together two simple numbers and display these. The "simplicity" of the numbers arises from the fact that they were only single digits and that their answer gave no carries. When larger numbers are added and carries arise, the 6502 handles them by use of its carry or C flag.

This is set automatically when an addition operation is carried out that brings about a carry between the two bytes of a two byte number - clear, eh? If not, please read on!

Using one byte, it's only possible to count up to 255, thus if we wish to count beyond this we have to use two bytes. These 16 bits then allow us to count up to 65535. If you have not yet read the section on binary and hexadecimal (Appendix 2) then it would be well to do it now! Naturally it would be possible to hold as large a number as there are bytes free; however, we will now consider the use of two bytes which is rather fancily described as DOUBLE PRECISION operation.

If two or more bytes are to be successfully utilised to represent a single number then they must be linked from the first to the second byte by some mechanism. This is the function that the carry flag performs and its operation is tested by the instruction:-

BCC	Branch on Carry Clear.

This tests for the carry flag being clear, i.e. set to 0, and executes a branch if this is so. One precaution is always best observed when testing for this flag, however. That is to ensure that the flag is in the expected state prior to the operation that may modify it. The instruction which performs this task is:-

CLC	CLear the Carry flag.

This sets the carry flag to 'clear' or '0' and is used prior to the process that may reset it in Program 4.1:

PROGRAM 4.1

```
        CLC
        LDA #0
        ADC #1
        BCC 771
        STA 1024
        RTS
```

NOTE:
As you are well aware by now, the assembler must be given a start address, i.e."START ADDRESS? From this chapter onwards this first line is omitted in the listing but the question must be answered.

When run, this program progressively increases the accumulator contents by 1 until at 255 the ADC #1 instruction flips the eight '1's over to eight '0's and sets the carry flag to 1. Thus when the accumulator is displayed with the instruction STA 1024, it is seen to contain '0' (i.e. an inverse @ on the screen).

The 6502 has a second test instruction for the carry flag, this being:-

BCS	Branch on Carry Set.

This tests for the carry flag being 'set', i.e. containing a '1', and if the test is positive, executes a branch. Program 4.2 illustrates this instruction in use:

PROGRAM 4.2

```
        CLC
        LDA #0
        ADC #1
        BCS 778
        JMP 771
        STA 1024
        RTS
```

Once again, this program progressively fills all eight bits of the accumulator with '1's and finally, on flipping these over to '0's sets the carry flag and terminates the program. At the end, the accumulator contains all '0's and therefore an inverse '@' is displayed on the screen.

Let's have a go at adding together two numbers larger than 256; we'll take 1257. First we have to calculate the MSB and LSB and to do this, of course, we have to convert the number to a hexadecimal format. Thus:-

```
INT (1257/4096)       = 0  therefore Right-most character = 0
INT (1257/256)        = 4  therefore 2nd character       = 4
INT ((1257-4x256)/16) = 8  therefore 3rd character       = 8
(1257-4x256-8x16)     = 5  therefore 4th character       = 5
```

Thus:

$1257_{10} = 048516$

and its two parts

MOST SIGNIFICANT BYTE	LEAST SIGNIFICANT BYTE
04_{16}	85_{16}

To add together two 1257_{10}'s we must first add the LSB's, check whether there is a carry and then add the MSB's taking into account the necessity (or otherwise) of a carry, i.e. first we add LSB's:

Hex	Decimal
85	85
+85	+85
carry+0A	16,10 = carry+0A

Let's look at that decimal addition a bit closer: we have said 5+5=10, and 8+8=16. The sixteen signifies a carry - had the answer been, say '15,10', then no carry would be required as $15_{10}=F_{16}$. However, 16 is not represented as G, but as 10: the carry being converted to the 1.

Then we add the MSB's:

```
 04
+04
 08
```

Next add in the carry

08 + carry = 09

In the explanation we have glibly said "+carry" and it is this operation that the C-flag does for the programmer. The flag is set to a '1' when an operation is carried out that leads to a "carry". The next operation that is carried out then takes account of this carry and adds 1 on to the next "add" carried out.

With C flag = "0"	With C flag = "1"
04 + 04 = 08	04 + 04 = 09

Thus the answer to the example is, in Hex 090A, or in decimal $9x256+10 = 2314_{10}$.

Now let's try doing that the long way - using the computer!

We can rely on the 6502 handling the carry but we can't rely on it knowing when to execute a carry! All double precision work is carried out least significant byte (LSB) first as it is during this addition operation that the carry arises and it is then stored ready for the most significant byte (MSB) addition. You may remember that when the 6502 is using indirect addressing commands it stores the LSB of the address first and the MSB second - this is the order that they are used when the index is added to the 'pointer' address. We could, if we so wished, stick to this organisation ourselves when we are storing 'numbers' (as distinct from addresses). But since we are doing the organising of the way in which the two bytes are added (not the 6502), there is no real advantage in storing 'numbers' either way (BASIC stores it's integers MSB followed by LSB).

In order to ensure that the LSB addition is not upset by a carry, it is most important that we preface the addition of the LSB's by the clear carry (CLC) instruction.

First we must work out the value of MSB and LSB in decimal, as both methods of putting data into memory that we have discussed so far. The decimal of the LSB of $485_{16}(1257_{10})$, i.e. 85_{16}, is

$$8 \times 16 + 5 = 133_{10}$$

and for MSB it is

$$0 \times 16 + 4 = 4_{10}$$

Now to write the program - but before we do that, let's just introduce a new instruction:-

NOP	No OPeration

When the 6502 meets it, it does nothing for two cycles of its operation.

We'll see why we put it in shortly; for now, type in the program.

PROGRAM 4.3

```
CLC                     Clear carry flag.
CLD                     Explained later.
LDA #133                Load A with LSB.
ADC #133                Add with carry 2nd LSB.
STA 1026                Store sum of LSB's in 1026.
NOP                     Do nothing.
LDA #4                  Load A with MSB.
ADC #4                  Add with carry 2nd MSB.
STA 1024                Store sum of MSB's in 1024.
RTS                     Return from machine code
                        subroutine.
END
```

Having done this you may run the program; it should put an inverse I J on the screen.

The output means: I or 9_{10} and J or 10_{10}

i.e. $90A_{16} = 231_{10}$

Stepping through that program, the various stages are:
(? signifies Random value.)

Stage	Accumulator	1026	1024	C Flag
START 828	?	0	0	?
CLC	?	0	0	0
CLD	?	0	0	0
LDA #133	133	0	0	0
ADC #133	10	0	0	1
STA 1026	10	10	0	1
NOP	10	10	0	1
LDA #4	4	10	0	1
ADC #4	9	10	0	0
STA 1024	9	10	9	0
RTS	9	10	9	0

Fig. 4.1

As the above chart shows, at the ADC #133, a carry is generated, the C flag is set to '1' and this carry affects the subsequent ADC. The other thing to notice is that at the ADC #4 a further carry is not generated and the C flag is therefore cleared to '0'. If you want to check this you could replace the NOP command with CLC which would clear the flag after it has been set and notice that the answer that you get would be 'incorrect'.

This can be done by POKEing into 777 the code for CLC, i.e. 24:

PROGRAM 4.4

```
POKE 777,24
```

You have to access BASIC to POKE 777,24. To do this, get back to the Assembler Main Menu, select X (EXIT).
You will see the prompt

```
ARE YOU SURE (Y/N)?
```

Type Y and you will then see the Applesoft prompt"]".
Type in

```
POKE 777,24
```

press RETURN, then type in RUN (be sure the Assembler disk is in the disk drive!) You will be returned to the Assembler Main Menu. Select R to run, and enter the starting address 768.
Now, when run, Program 4.3 modified by 4.4 will give a display of:-

```
H J
```

In this run, the value of J has been computed and when its value of 266 overflowed, 256 was carried, the carry bit set and 10 was stored in the accumulator. However, this 256 was lost when the carry bit was later cleared by CLC (Program 4.4).

Using Hexadecimal Inputs

I'm willing to bet that you were somewhat horrified by the messy conversions into and out of hexadecimal which were carried out when preparing to write Program 4.3. However, the assembler possesses a few lines that save at least one of these tasks, the calculation from hexadecimal back to decimal. Thus, any operand can be entered in hexadecimal, providing it is preceded by a '$' sign. Utilising this, Program 4.3 becomes:

PROGRAM 4.3a

```
CLC
CLD
LDA #$85
ADC #$85
STA 1026
NOP
LDA #$04
ADC #$04
STA 1024
RTS
END
```

When Program 4.3a is run, it gives exactly the same result as Program 4.3, i.e. I J. Select the list option and you will see that the assembler has converted the $85 into a decimal value and POKEd this into memory.

EXERCISE 4.1

Using hex inputs, add together 1807_{16} and $2AFA_{16}$. Verify your program afterwards in base 10. Answer in Chapter 9.

To recap the use of the C flag; it stores the fact that an addition operation has yielded a carry. If the C flag is left set by mistake then this carry will be added to the next ADC operation whether you want it or not. For the carry to be passed from the LSB to the MSB you have to arrange the ADC's to be done in the appropriate order.

The 6502, of course, has an instruction which carries out subtraction with carry:

SBC	SuBtract from the accumulator with Carry, the data at the specified memory location.

e.g. SBC 891

means look in memory location 891 and subtract the number that you find there from the number in the accumulator.

However, in the same way as it was necessary to prepare the carry flag for addition by clearing it to '0', it is also necessary to prepare it for subtraction. Not unexpectedly, though, it is necessary to set it, rather than clear it i.e. to '1' rather than to '0'. The instruction which sets the carry flag is:

SEC	SEt the Carry bit to '1'.

Let's look at that in a simple program to take 2 from 4. For a change we will use the immediate mode to load the data.

PROGRAM 4.5

```
        SEC
        LDA #4
        SBC #2
        STA 1024
        RTS
        END
```

By now you should be well enough equipped to try a double precision subtraction on your own so have a go at the exercises below - in case you have a problem the answer to Exercise 4.1 is explained in fair detail to demonstrate the carry operation.

EXERCISE 4.2

Write a program to subtract 600 from 800 using ABSOLUTE ADDRESSING. Store data in memory locations 890 onwards. Display the answer in 1034.
Answer in Chapter 9.

EXERCISE 4.3

Write a program to subtract 500_{10} from the sum of 300_{10} and 40010.

Display the answer in 1040/1 in the order MSB/LSB.

Answer in Chapter 9.

Multiplication

The arithmetic instructions available for the 6502 allow additions and subtractions to be carried out but no provision is made for multiplication. This has to be carried out, therefore, by a series of repeated additions. For example, the sum 2x3 can be expressed as 2+2+2, and is thus relatively simple to evaluate. The process is thus one of adding 2 to the accumulator three times and requires the three to be set up in a loop to define the number of passes. Of course, the accumulator should contain zero before we do the adding.

The process is illustrated in Fig. 4.2 below, where Y = number of passes through the loop (in this case 3), and A = current sum.

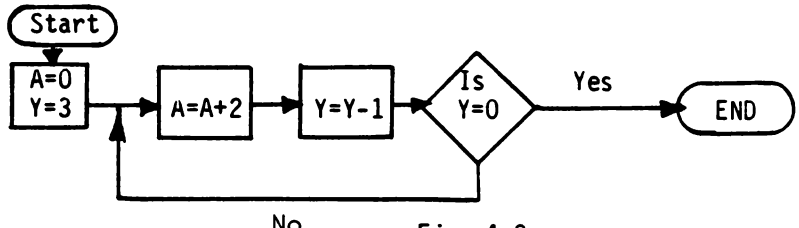


Fig. 4.2

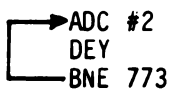
It is put into Program 4.5 below:

PROGRAM 4.5

```
        CLC
        LDY #3
        LDA #0
        ADC #2
        DEY
        BNE 773
        STA 1024
        RTS
```

When run, the program gives an inverse 'F' (an APPLE '6') in 1024. Within this program, the key segment is

PROGRAM 4.5a



This small loop which does the work is known as an ALGORITHM. One limitation of this simple algorithm is that it can only handle an answer up to 255 - after that, it generates a carry and clocks the accumulator back to zero. If no account is taken of this carry, then the answer loses 256 for each loop! When elaborated somewhat, this algorithm can handle double precision multiplication. This can be achieved by checking for a carry after each addition and if a carry is generated, adding one onto the MSB. One way of carrying out this incrementing is by means of the instruction:-

INC	INCrement the contents of the specified memory location.

Program 4.6 shows the algorithm 4.5a elaborated to record the number of carries generated and to increment the MSB.

PROGRAM 4.6

```
        LDY #0      } Set up 905 to receive carry bits.
        STY 905     }
        LDY #17       Set up number of times around loop.
        LDA #0        Clear accumulator to receive sum of LSB's.
    ┌──►CLC         Clear carry prior to adding.
    │   ADC #16     Add 16 to A once.
    │  ┌►BCC 785     If no carry generated then skip...
    │  │ INC 905     Increment carry record.
    │  └►DEY
    └─── BNE 777     Check if Y decremented to zero.
        STA 1026      Display LSB sum.
        LDA 905     }
        STA 1024    } Display MSB.
        RTS         }
```

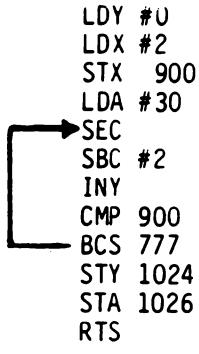
When run, this should display inverse A P or 256+16, i.e. 16x17.

In the example used - 16x17 - it made little difference whether the algorithm was arranged to add 16 seventeen times or to add 17 sixteen times. If the sum 2x100 was attempted, however, then it would obviously be much quicker to add 100 twice rather than to add 2 one hundred times. This could be tackled by writing a short subroutine that ensures that the multiplier is provided with the smallest of the two values, and thus the algorithm is cycled the least possible number of times. However, on page 4-22 another method of multiplication will be introduced - binary multiplication - and this keeps the number or iterations down to the minimum.

Division

Division using the 6502 is a process of repeated subtraction in the same way that multiplication is one of repeated addition. It is illustrated in Program 4.6a in which 30 is divided by 2. In this the accumulator is used to store the running remainder, i.e. starts with 30 and progressively declines to zero (30,28,26...4,2,0). The X register is used to load the divisor into memory while the Y register counts the number of times that the subtraction can be made.

PROGRAM 4.6a



When run, this displays the quotient 15 (as an inverse letter O) in 1024 and the remainder 0 (as an inverse @) in 1026.

Binary Coded Decimal Arithmetic

In addition to numbers being represented in binary and decimal notation, a hybrid or mixed notation, binary coded decimal or BCD, also exists. The usage and format of this is described in Appendix 2. BCD forms a bridge between the two notations and in many cases greatly facilitates output. Fortunately for us, the 6502 chip can handle BCD arithmetic and is turned on to the BCD handling mode by the instruction:-

SED	SEt Decimal mode of operation.

This instruction sets the D flag automatically to a '1' and thereafter arithmetic is done in BCD. When the decimal mode of operation is no longer required it is cleared with the instruction:-

CLD	CLear the Decimal flag.

This sets the flag back to a '0' and thereafter arithmetic is done in binary. You may remember that we have used the CLD instruction on a number of occasions, prior to using the ADC and SBC instructions. It should be clear to you now that we were ensuring that the 6502 carried out binary arithmetic.

A simple program to add together 1 and 2 using BCD is given in Program 4.7.

When run, Program 4.7 puts a 'C' in 1024. It is usually considered to be good practise to clear the decimal flag after doing any BCD arithmetic, since most arithmetic is done in binary - hence the CLD just before the RTS.

The example given in Program 4.7 is identical in effect to the arithmetic that we have done to date, with the exception that in BCD the carry occurs after each half-byte exceeds 9. This is demonstrated below in Program 4.8, which adds together two 6's. If Program 4.7 is still in 768 then 4.8 can be POKEd in by:

```
POKE 771,6
POKE 776,6
```

See page 4.6 for instructions for accessing BASIC to do the POKEs.

PROGRAM 4.7

```
SED
CLC
LDA #2
STA 900
LDA #1
ADC 900
STA 1024
CLD
RTS
```

PROGRAM 4.8

```
SED
CLC
LDA #6
STA 900
LDA #6
ADC 900
STA 1024
CLD
RTS
```

When run, Program 4.8 puts an inverse R in 1024, which is its way of saying 12! This comes from the way that BCD is stored in memory as NYBBLES (a nybble is half a byte - ouch!! - don't blame me, I didn't invent the word). The 'R' itself comes from the APPLE's poke code of 18, which in binary is:

$$18_{10} = 00010010_2$$

However, the memory location is storing two nybbles rather than one byte and hence:

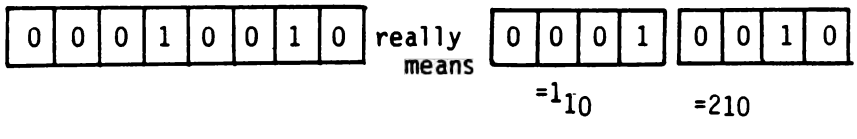


Fig. 4.3

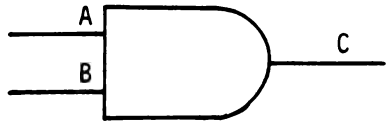
i.e. the number represents 1x10+2x1, or 12_{10}.

The above example emphasises the problems that arise when thinking in decimal and working in binary. As the 6502 itself works in binary and won't easily be persuaded to change, it is necessary to master the techniques of binary number processing.

If we look at the answer to Program 4.8, which consisted of two nybbles held within one byte, we see one of the problems of bit manipulation. What was needed in this case was a technique for modifying individual bits within a byte. In order to extract the lower order nybble from the binary number it is only necessary to erase the higher order nybble, i.e. to fill it with '0's. This can be done with an instruction:-

AND	Perform a logical AND between the accumulator

An AND is a logical operator that compares two logic states and produces an output based on the comparison. If we examine a logic AND gate such as is used in electronic circuitry, it makes the AND function clearer.



An Electronic AND

Fig. 4.4

Figure 4.4 shows an AND gate with two inputs A and B, and an output C. Its function is such that if both inputs, A AND B, are set at '1', then its output, C, is a '1'. However, if either or both of its inputs A AND B are '0' then its output is '0'.

This is normally expressed in what is known as a TRUTH TABLE, that for the AND gate in fig. 4.4 being shown in fig. 4.5.

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Truth Table for Electronic AND

Fig. 4.5

To use the table, the value of C - the output - is read off for the appropriate inputs of A and B. Thus, taking an A input of '0' and a B input of '0' the output C is 0.

EXERCISE 4.4

Using the truth table, fig. 4.5, find the logic output (C) obtained for the following inputs:-

A=1 AND B=0
A=0 AND B=1
A=1 AND B=1

Answer in Chapter 9.

When an AND is performed by the 6502, it operates on all eight bits in the accumulator simultaneously. Thus if 255 is ANDed with 1 then:

255_{10} = 11111111_2

1_{10} = 00000001_2

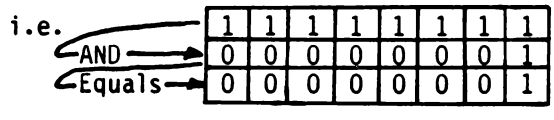


Fig. 4.6

The result of the operation is that all the bits ANDed with a '0' have been stripped off leaving only the first bit.

EXERCISE 4.5

What is the result when 149_{10} is ANDed with 52_{10}?

If you have problems, work it out bit by bit using the truth table.

Answer in Chapter 9.

As the above exercise shows, the AND instruction can be used to strip BITS from a number and could be used to convert part of the BCD 12 from Program 4.8. This BCD '12' was stored as two nybbles in one byte. If the Most Significant Nybble (MSN) could be changed into four zeros then the byte would read out directly as the value of the Least Significant Nybble. Such a masking out of bits can be done by using an AND command, as any '1' in the ANDing number will leave '1's in the number ANDed as they were while '0's in the ANDing number will switch any '1's in the ANDed number to '0's.

Let's try that with the BCD 12.

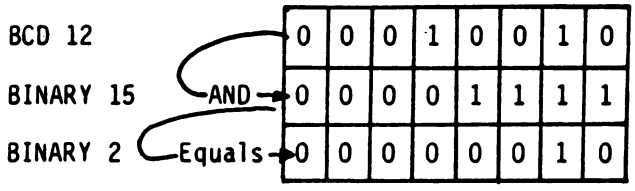


Fig. 4.7

By ANDing the BCD number with 00001111_2 (binary 1510), the four most significant bits have been erased and the number converted into the LSN (in this case 2_{10}).

In a program the AND instruction may be used with several different addressing modes, the absolute mode being illustrated in:

PROGRAM 4.9

```
LDX #15      Load X with '15'.
STX 900      Store X in 900.
LDA #18      Load A with '18'.
AND 900      AND A with 900.
STA 1024     Store A in 1024.
RTS
```

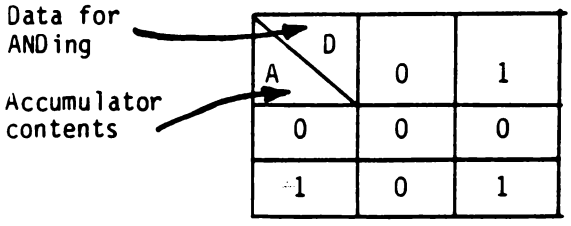
When run, this will display an inverse 'B' in 1024.

Using immediate addressing, AND #, Program 4.9 can be re-written as below:

PROGRAM 4.9a

```
LDA #18
AND #15
STA 1024
RTS
```

When defining the truth table of the AND instruction it is written somewhat differently from that used in electronics. This is because the result of the AND operation is deposited back in the accumulator, i.e. this forms both part of the input and the output. The truth table for AND is given in figure 4.8 below:



Truth table for AND

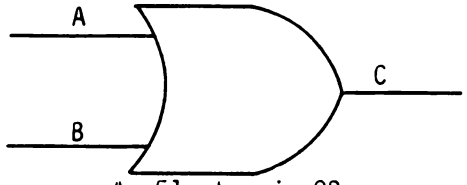
Fig. 4.8

The 6502 chip also uses two other logical operators, one of these being an OR function.

It is defined formally as:-

ORA	Perform a logical inclusive OR between the Accumulator and the data specified.

In electronic circuitry, the OR is depicted as shown in Fig. 4.9.



An Electronic OR

Fig. 4.9

Its mode of operation is that if a '1' is present on A OR B then the output C is set to a '1'. This is a bit like an AND in reverse - the AND gives a '1' only if both inputs are '1', while the OR gives a '0' only if both inputs are '0'. The truth table for the 6502's ORA command is given below on figure 4.10.

A \ D	0	1
0	0	1
1	1	1

Truth table for OR

Fig. 4.10

In action, the ORA command has the following effect:

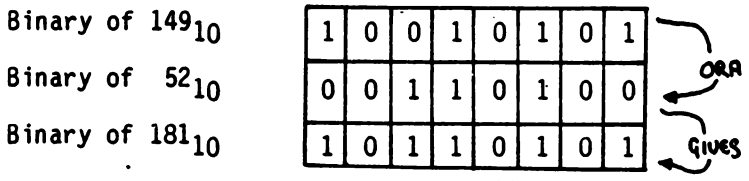


Fig. 4.11

Putting that into a program gives:

PROGRAM 4.10

```
LDA #149
ORA #52
STA  1024
RTS
```

When run, this program will put a 181 (a 5) in 1024.

As with AND, ORA has several modes of addressing to suit different applications.

Third among the logical operators is:-

EOR	Perform a logical Exclusive OR between the accumulator and the data specified.

This operation is probably the least easy to understand and is best illustrated by means of the truth table, fig. 4.10.

A \ D	0	1
0	0	1
1	1	0

Truth table for EOR

Fig. 4.12

One way of expressing the function is that the output will be '1' if either of the inputs is '1' but not both. Using this instruction with the above example, i.e. 149_{10} EORed with 52:

Binary of 149_{10}

Binary of 52_{10}

Binary of 161_{10}

Binary of 149_{10}	1	0	0	1	0	1	0	1
Binary of 52_{10}	0	0	1	1	0	1	0	0
Binary of 161_{10}	1	0	1	0	0	0	0	1

EOR

GIVES

Fig. 4.13

The program to demonstrate that is below:

PROGRAM 4.11

```
LDA #149
EOR #52
STA  1024
RTS
```

When run, the program will put a 161 (a "!") in 1024.

EOR, like the other logical operators, has several modes of address to facilitate its use in programs.

EXERCISE 4.6

Calculate the result of the following logical operations:-

(i) 100_{10} ANDed with 87_{10}.

(ii) 75_{10} ORed with 27_{10}.

(iii) 99_{10} EORed with 57_{10}.

(iv) 94_{10} EORed with the result of 100_{10} ANDed with 87_{10}.

Write a program to verify each operation.

Answers in Chapter 9.

Other Forms of Bit Manipulation

Other 6502 instructions exist that enable one to manipulate bits within a byte and as a group these lead to the movement of bits to the right or left within the byte itself.

In the earlier example using BCD arithmetic, the AND instruction was able to isolate the LSN from the byte. It was not possible, though, to extract the MSN using the available logic. This does become possible, however, using one of the bit manipulation commands:-

LSR	Logical Shift of the specified contents one bit to the Right.

When this is done, the bits are moved along one place to the right with the right-most bit falling off into the carry and a '0' being put into the left-most bit. Thus the LSR command operating on 149_{10} gives the following:

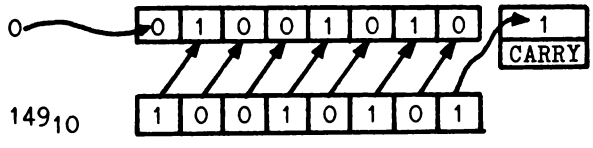


Fig. 4.14

As with most 6502 commands, LSR has several addressing modes and the particular address used informs the 6502 where the data which is to be shifted resides. Thus:

```
LSR       means logical shift right of the data in A.
LSR 900   means logical shift right of the data in 900.
```

Using the Accumulator mode LSR to prove the above example is:

PROGRAM 4.12

```
          LDA #149
          LSR
          STA 1024
          RTS
```

When run, this will put a 74_{10} (a flashing 'J' in 1024)

By using four such right shifts the MSN nybble in a byte can be moved into the place of the least significant nybble and the left-most four bits filled with 0's. This enables one to isolate the MSN in a BCD calculation.

This is demonstrated in Program 4.13, this time using LSR in its absolute mode.

PROGRAM 4.13

```
          LDY #18   }  Load '18' into 900.
          STY 900   }
          LDY #4       Set loop counter.
       ┌─►LSR 900  }  Shift 900 four
       │  DEY      }  places to
       └──BNE 775  }  right.
          LDA 900   }  Print out contents
          STA 1024  }  of 900
          RTS
```

When run, the program will print a 1 (an inverse 'A') in 1024.

EXERCISE 4.7

Suppose that the answer to a problem in BCD is 86_{10}.

Write a machine-code program to decode this and display the answer in decimal (POKE form, i.e. characters) in 1024 and 1025.

A possible answer in Chapter 9.

A further 6502 instruction mirrors LSR in that it moves the bits to the left; it is:-

ASL	Arithmetic Shift Left: Shift the specified contents one bit to the left.

The left-most bit is shifted into the carry bit and bit 0 is loaded with a '0'. An ASL instruction, operating on 149 gives:-

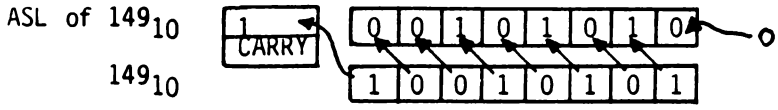


Fig. 4.15

Using the various modes of addressing, ASL can operate on data in different locations, e.g.

ASL means Arithmetic Shift Left on data in A.
ASL 900 means Arithmetic Shift Left on data in 900.

Using the accumulator mode to test the above example is:

PROGRAM 4.14

```
LDA #149
ASL
STA 1024
RTS
```

When run, this puts a 42 (an inverse asterisk) in 1024.

Binary Multiplication

We have seen from programs 4.5 and 4.6 that multiplication can be carried out using a repetitive, or RE-ITERATIVE, process but we have also seen that this is a lengthy process and at times very tricky. However, we tackled the problem very much from a point of view of conventional arithmetical processes and binary has its own way of doing these things! As the 6502 itself thinks in binary and has a number of instructions for manipulating the bits within its bytes, binary arithmetic has a lot to offer.

First, let's examine our way of doing decimal (conventional) arithmetic. Take the sum 13x14. We define 13 as the MULTIPLICAND and 14 as the MULTIPLIER and lay the multiplication out as below:-

```
  13      Multiplicand
X 14      Multiplier
----
  52
 130
----
 182
```

Fig. 4.16

In this conventional format, we first multiply the multiplicand by the lowest digit of the multiplier and store this as the first partial product, i.e. 4x13=52. Next, we multiply the multiplicand by the second digit of the multiplier, i.e. 1x13, and then multiply this by 10 to obtain the second partial product, i.e. 13x10=130. Thus the total answer is the sum of the two parts, i.e. 52+130=182. Actually, in the second stage it would have been more correct to say that we multiplied the multiplicand by the second digit of the multiplier and then multiplied the result by the BASE (which happened to be 10). The total answer was then the sum of the two partial products.

It is quite possible to perform the same multiplication process using numbers in binary format.

For example, to multiply 5 x 7 in binary,

5_{10} = 0101 and 710 = 0111 (working only to 4 bits)

```
i.e.   7          0111
     x 5    =   x 0101

Partial Product 1     0111    Current total 1 =   0111
Partial Product 2    00000    Current total 2 =   0111
Partial Product 3   011100    Current total 3 = 011011
Partial Product 4  0000000    Current total 4 = 011011

          ANS  =   100011  = 32 + 2 + 1 + 35(10)
```

Fig. 4.17

With all digits in the multiplier equal to one, all the significant bits in the partial products parts have the same pattern of digits as the multiplicand, i.e. "111". Thus the multiplication process in binary reduces to one of successive addition following movement to the left of multiplicand.

8-Bit Multiplication

The flow diagram for this process is given in figure 4.18, where Answer=ANS, D=Multiplicand, R=Multiplier, N=current bit number, LSB=least significant bit of multiplier.

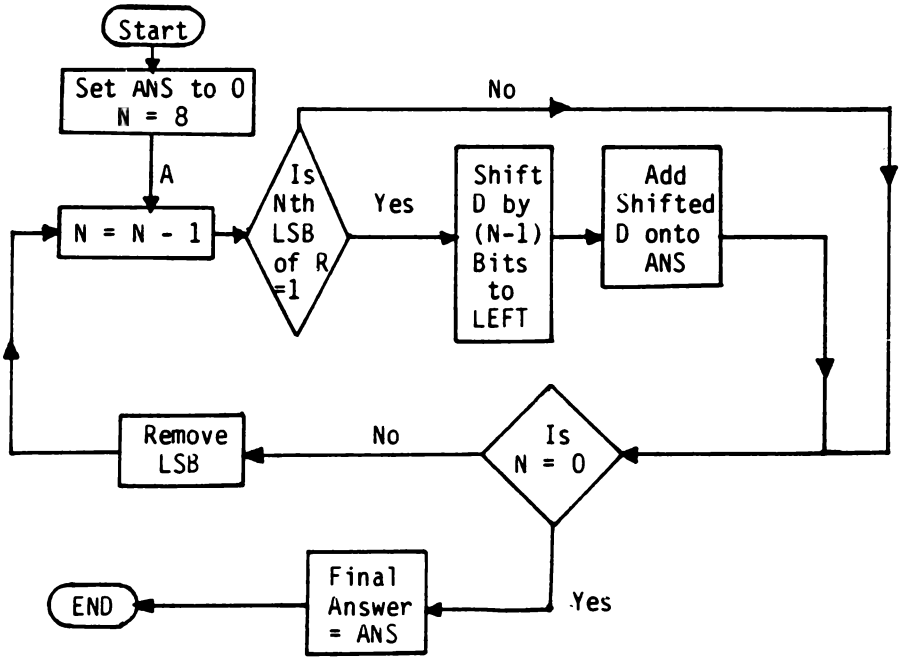


Fig. 4.18

Working through this flow diagram for the example of $2_{10} \times 210$

$$D = R = 00000010_2$$

Successive passes past A are referred to as A1, A2, etc. For this very simple example with no carries, only four passes through the loop will be made (two would suffice as only the two right-most digits are significant.

START	ANS = 0	N = 0	
A1	N = 1	1st LSB = 0	N = 8
A2	N = 2	2nd LSB = 1	
	Shift R (N-1) bits (i.e. 1) to left, i.e. $0010_2 \rightarrow 0100_2$		
	add R to ANS, i.e. ANS=0+0100_2=0100_2		
	N = 8		
A3	N = 3	3rd LSB = 0	N = 8
A4	N = 4	4th LSB = 0	N = 8

The process then carries on uneventfully for the remaining four zero bits, adding nothing to the ANS.

Thus the answer = $0100_2 = 4$
Putting this into a program gives:

PROGRAM 4.15

```
        LDX #2          }
        LDY #8          }
        STX 901         } Initialisation
        STX 902         }
        LDA #0          }

        LSR 902         } Check whether multiplication necessary.
        BCC 789         }

        CLC             } 'Multiply' if necessary.
        ADC 901         }

        ASL 901         }
        DEY             } Check if all bits processed.
        BNE 780         }

        STA 1024
        RTS
```

When run, the program will print an inverse D (4) in 1024. It may be checked by changing the LDX # instruction to input a different multiplicand and multiplier.

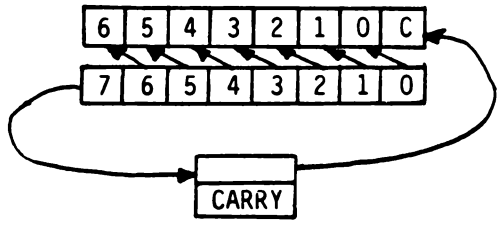
> EXERCISE 4.8
>
> Re-write Program 4.15 so as to multiply two different numbers together.
> One possible answer in Chapter 9.

Unfortunately, Program 4.15 is only really half true as an eight bit multiplication routine and works only over a range of small numbers for the multiplier and multiplicand. In the full routine, the ASL instruction multiplies the multiplicand by the base eight times. Thus, by the eighth shift the right-most bit would have fallen off the left-hand end of the register. In fact, with the number 2_{10} worked through in the example, the second bit (representing 21, or 2_{10}) would be lost after seven shifts. However, this did not affect the overall result because after two LSR's of the 00000010, all the "1"s had been cleared and all subsequent partial sums were equal to zero.

Were Program 4.15 to be used with a larger number where the final answer involved a carry (i.e. was greater than 255), then this loss of left-most bits would be significant and the answer obtained would be wrong. Fortunately, when the left-most bit falls out of a register during an ASL it does not just fall into space but is caught in the carry. Thus the problem is one of retrieving this and transferring it to the MSB of the answer. This can be achieved by means of the command:-

ROL	ROtate Left the contents of the specified address.

In this operation, all the bits of the specified address are rotated left, with the carry bit being loaded into the right-most bit and the left-most bit being transferred to the carry.



The ROL Operation

Fig. 4.19

As the rotation involves the carry bit, it is known as a 9-bit rotation and provides a means of picking the carry bit back up again. Using ROL enables one to write an eight bit multiplication accommodating the carry but produces a rather complex program. Such a program is listed in Chapter 5 (page 5-5) as, to make it clearer to follow, both labels and memory labels have been used.

The instruction ROL has a right-handed colleague:-

ROR	ROtate Right the contents of the specified address.

Both these instructions can be addressed in several ways and take forms as:-

ROL	ROtate Left the contents of the Accumulator.
ROR	ROtate Right the contents of the Accumulator.

Such forms will be described when utilised.

One other instruction is available for bit manipulation:-

BIT	AND specified content's BITs with accumulator.

Thus, BIT 900 performs a logical AND between the bits in the specified memory location, i.e. 900, and the accumulator.

While BIT performs the same logical function as AND, it differs in that it leaves both accumulator and memory as they are. It does, however, modify the relevant flags in the PSW in the following way:

- The Z flag is set if the result of the ANDing is a zero (and cleared, of course, if the result is not zero).

- The N flag is affected as follows:- bit 7 of the location being tested is copied to the Processor Status register (bit 7 of the PSR being the N flag). This is a very convenient way of testing whether the contents of a particular location are positive or negative without the necessity of loading the value into one of the registers.

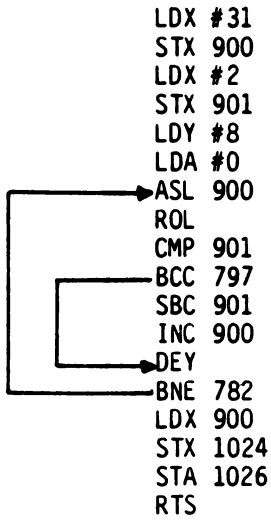
- The V flag (which we haven't really discussed yet) is bit 6 of the Processor Status register. The BIT instruction also copies bit 6 of the location being tested to bit 6 of the PSR. This isn't quite so useful as the N flag as bit 6 doesn't normally signify anything very special. However, if you look at some of the clever machine code programming used in the BASIC interpreter and operating system, you will occasionally find some very neat uses of the BIT instruction operating on the V flag.

Using these binary instructions, a process analogous to binary multiplication can be carried out.

8-Bit Binary Division

This process is analogous to the binary multiplication routine, needing only 8 re-iterations to handle an 8-bit number. It is illustrated in Program 4.15A, where the dividend (in this case 31) is stored in location 900 and the divisor (2), in 901. The Y register is used as the loop counter to ensure that 8 passes are made through the algorithm. By means of an ASL and a ROL instruction, the remainder is built up in the accumulator.

PROGRAM 4.15a



```
        LDX #31
        STX 900
        LDX #2
        STX 901
        LDY #8
        LDA #0
        ASL 900
        ROL
        CMP 901
        BCC 797
        SBC 901
        INC 900
        DEY
        BNE 782
        LDX 900
        STX 1024
        STA 1026
        RTS
```

When run, this will display the quotient 15 (as an inverse O) in 1024 and the remainder 1 (as an inverse A) in 1026.

CHAPTER 5

Advanced Functions of the Assembler

Labels

THE USE OF LABELS ENABLES A PROGRAM TO BE DIRECTED TO A NAMED instruction without the necessity of calculating branches or jump addresses. A fancier term for label is SYMBOLIC LABEL as the label itself is symbolic of a location in memory. For instance, the instruction

```
BNE LOOP1
```

instructs the assembler to create the machine code that tells the 6502 to branch to an instruction labelled LOOP1. Thus, LOOP1 STAX 1024 creates a label called LOOP1, whose address is the same as the "STAX" in STAX 1024. In order to tell the assembler that a label is a label and not an instruction, it is preceded by an asterisk (*). This rule is only a convention that has been chosen when writing this particular assembler. Thus, the beginning of LOOP1 would be entered:

```
*LOOP1 STAX 1024
```

Further conventions must be observed when using labels, particularly those concerning spaces. The asterisk, for instance, must be followed immediately by the label (no space between). It may be as long as required but must NOT contain any spaces. There is no particularly technical reason for this, it is simply that the assembler looks for a space in order to work out how long the label is. For this reason the label must be followed by a space and then a normal instruction. These rules may sound a little formidable but don't worry, the assembler will pick up any errors and let you know what is wrong with any particular line. When referring to a label in an instruction it is only necessary to replace the operand with the label itself.

To summarise:

(i) A label is defined by the asterisk (*) that precedes it.

(ii) The label may be of any length but it is as well to stick to about six characters.

(iii) There must be no gap between the asterisk and the label.

(iv) The label must be followed by a gap prior to the instruction.

It all sounds a little complicated so let's see it illustrated in a program. This uses two loops, called "LOOP1" and "LOOP2" and does some unnecessary branching and jumping by way of illustration.

PROGRAM 5.1(a)

```
        LDX #160
        JMP LOOP2
*LOOP1  LDA #165
        STA 1183,X
        DEX
        BNE LOOP1
        JMP END
*LOOP2  LDA #166
        STA 1023,X
        DEX
        BNE LOOP2
        LDX #120
        JMP LOOP1
*END    RTS
        END
```

Although this program jumps about somewhat, it is still relatively easy to follow. It starts by initialising X then jumps to loop 2 and on completion reinitialises X and then jumps back to loop 1, and from there to the END. Note that a *END will not indicate the end of assembly; the "*END" is a label whereas "END" on its own (without the asterisk) is the pseudo-code that terminates the assembly process. Once this process is complete, the program will reside in memory in exactly the same format as any other program that has been entered. To check this, list the program - from 768 - and the following should appear:

PROGRAM 5.1(b)

	In Assembly Language	In disassembled Assembly Language
	LDX #160	LDX #160
	JMP LOOP2	JMP 784
*LOOP1	LDA #165	LDA #165
	STA 1183,X	STA 1183,X
	DEX	DEX
	BNE LOOP1	BNE 773
	JMP END	
*LOOP2	LDA #166	JMP 797
	STA 1023,X	LDA #166
	DEX	STA 1023,X
	BNE LOOP2	DEX
	LDX #120	BNE 784
	JMP LOOP1	LDX #120
*END	RTS	JMP 773
	END	RTS

Program 5.1(b) can exist in a variety of forms, three of which are readily available. In its original form it was written in assembly language with labels and this was converted by the assembler into machine code and stored in memory in this form. When the assembler is then asked to list this program it reads the machine code from memory and changes this back into assembly language. Were this process to be carried out immediately after assembly then it would be possible to re-label the label points by editing the assembler but as currently written this is not so, in common with other assemblers. Moreover, once the BASIC program has been re-run, the variables, i.e. the LABELS and LABEL REFERENCES, will have been lost. Re-creating the assembly program from machine code is known as DISASSEMBLY, i.e. the 'list' command could be re-titled as 'disassemble' and this process cannot re-create labels.

When run, this program will print 7 lines of characters altogether, at the top of the screen, one third and two thirds of the way down the screen due to the mapping of the screen into RAM. The characters will be %'s and &'s.

EXERCISE 5.1

Add a further loop - loop 3 - after loop 2 in program 5.2. Re-write the program to run loop 3 first, followed by loop 1 and then loop 2. Loop 3 should put about two rows of asterisks on the screen below the percentage signs.

A possible answer is given in Chapter 9.

Memory Labels

In addition to labelling instructions, the assembler also allows memory locations to be given labels. Once again, the assembler needs to be told what to expect and the presence of a memory label is indicated by an "@" at the beginning of line. It is followed immediately by the name assigned to that location and then, after a space, by the location itself in decimal. Thus the instruction

```
@LSB 900
```

informs the assembler that memory location 900 may, in the rest of the program, be referred to as "LSB".

Program 5.2 illustrates the use of memory labels in double precision addition - it adds together two 16 bit numbers:-

Number 1 = 2760_{10}
Number 2 = 948_{10}

made up of LSB1 and MSB1, and LSB2 and MSB2; the answers are stored in ANSLSB and ANSMSB.

PROGRAM 5.2

```
@LSB1 900       Define memory location for LSB1.
@MSB1 901       Define memory location for MSB1.
@LSB2 902       Define memory location for LSB2.
@MSB2 903       Define memory location for MSB2.
@ANSLSB 904     Define memory location for ANS LSB.
@ANSMSB 905     Define memory location for ANS MSB.
```

```
LDA #10        } Store MSB1 in memory.
STA MSB1       }
LDA #200       } Store LSB1 in memory.
STA LSB1       }
LDA #3         } Store MSB2 in memory.
STA MSB2       }
LDA #180       } Store LSB2 in memory.
STA LSB2       }

CLC              Clear carry prior to addition.

ADC LSB1       } Add LSB's together, store answer
STA ANSLSB     } in ANSLSB and print on screen.
STA 1025       }

LDA MSB1       } Load MSB1 and add with carry to
ADC MSB2       } MSB2, store answer in ANSMSB and
STA ANSMSB     } print on screen.
STA 1024       }

RTS
END
```

When run, Program 5.2 will display an inverse N (14) in 1024 and a flashing '<' (124) in 1025.

As promised (!) in Chapter 4, a listing is given below of an eight bit binary multiplication using labels and memory labels. In order to illustrate the carry operation the first few loops through this program are illustrated on Fig. 5.1. The numbers 255 were chosen as multiplier and multiplicand as their binary pattern of eight ones is easy to follow. They also give an early carry, although any multiplier over 128 would have given the carry when ASLed.

Abbreviations used in this program are:

```
MPR     =  Multiplier   =  255
MPD     =  Multiplicand =  255
TEMP    =  Temporary location to store carry bit.
RESLSB  =  Result - Least significant bit.
RESMSB  =  Result - Most significant bit.
ALGO    =  Start of multiplication algorithm.
NOCARRY =  Jump to point if no carry arises.
```

PROGRAM 5.3

```
@MPR      900                 } Define memory location for
          LDA #255            } multiplier, load and store
          STA   MPR           } multiplier.
@MPD      904                 } Define memory location for
          LDA #255            } multiplicand, load and store
          STA   MPD           } multiplicand.
@TEMP     902                 } Define memory locations for
@RESLSB   906                 } temporary store, and LSB/MSB
@RESMSB   907                 } of result.
          LDA # 0             } Initialise (by loading in
          STA     TEMP        } zero) the temporary and
          STA     RESLSB      } result store.
          STA     RESMSB      }
          LDY # 8             Set loop counter to 8.
*ALGO     LSR     MPR         } Check if right-most bit of
          BCC     NOCARRY     } multiplier=0; branch if so.
          LDA     RESLSB      } Calculate current partial
          CLC                 } product and add in to
          ADC     MPD         } current partial sum.
          STA     RESLSB      }
          LDA     RESMSB      } Add current carry into MSB
          ADC     TEMP        } sum.
          STA     RESMSB      }
*NOCARRY  ASL     MPD         } Current Partial=0, set up
          ROL     TEMP        } next loop.
          DEY                 }
          BNE     ALGO        }
          LDA     RESLSB      }
          STA     1025        } Display result of LSB and
          LDA     RESMSB      } MSB
          STA     1024        }
          RTS                 }
```

When run, Program 5.3 will print a ~ in 1024 and an A in 1025, i.e. answer of 254,1 or $FE01 = 65025.

Fig. 5.1 below steps through the first few stages of this program, once all the registers are set up. The contents of each address are shown only when they change.

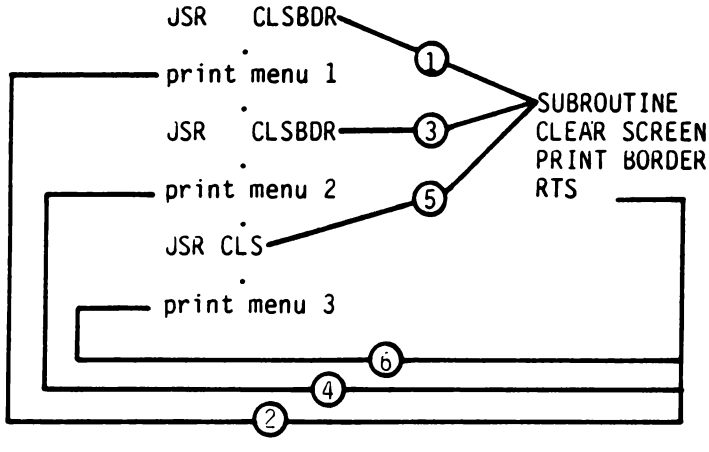
LABEL	INSTRUCTION	Y-REGISTER	ACCUMULATOR	MPR MULTIPLIER	C	TEMP	MPD	RESLSB	RESMSB
		0000 1000	0000 0000	1111 1111		0000 0000	1111 1111	0000 0000	0000 0000
ALGO	LSR MPR			0111 1111	1				
	BCC NOCARRY								
	LDA RESLSB		0000 0000						
	CLC				0				
	ADC MPD		1111 1111						
	STA RESLSB							1111 1111	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0000						
	STA RESMSB								0000 0000
NOCARRY	ASL MPD				1		1111 1110		
	ROL TEMP				0	0000 0001			
	DEY	0000 0111							
	BNE ALGO								
ALGO	LSR MPR			0011 1111	1				
	BCC NOCARRY								
	LDA RESLSB		1111 1111						
	CLC								
	ADC MPD		0000 0001		1				
	STA RESLSB							0000 0001	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0001						
	STA RESMSB								0000 0001
NOCARRY	ASL MPD						1111 1100		
	ROL TEMP				0	0000 0011			
	DEY	0000 0110							

Fig. 5.1

Macro Instructions

One further feature of your assembler is that it enables you to use MACRO instructions. These are blocks of code that you wish to repeat and are thus given a label. When you wish to insert these into your program you simply need to type in the label and this automatically inserts the whole macro. Thus a macro is very similar to a subroutine except that the assembler writes it in every time it is called rather than using JSR instructions. As an example, take a short routine that clears the screen and puts in a border called CLSBDR. Assuming that the program has a hierarchical menu structure and calls this routine 3 times, it could be written in as (i) below of Fig. 5.2 in which the routine is called as a subroutine, or as in (ii) in which the routine is written in 3 times as a macro.

(i)



(ii)

```
MACRO CLSBDR
 .
 MENU 1
 .
MACRO CLSBDR
 .
 MENU 2
 .
MACRO CLSBDR
 .
 MENU 3
```

Fig. 5.2

The above figure is somewhat unfair on subroutines in that they are not so messy as the picture suggests, the 6502 doing much of the organisation. Where macros score is in producing a program that reads more logically and is easier to follow. They take up more space than a subroutine, which is only written once, but as they don't use up processor time jumping about, they run more quickly. Just whether a macro or a subroutine is more preferable in any particular case is left to the individual programmer.

To identify a macro on this particular assembler it is preceded by a "+" sign (as the macro is later 'added' into the program). As with other features, the + sign must immediately precede the macro's name which may be any length and this should be the only entry on that particular line, e.g. "+MACRO1". The first occurrence of the macro is then typed in. The end of the macro is signalled by a "+" sign followed by "END". To include a further copy of the macro in the program at a later point, another MACRO1, say, just the line +MACROI needs to be entered into the program at the appropriate point - see Fig. 5.3.

```
  .
  .
+MACRO1

 LDA # 90        } First occurrence of MACRO1
 LDX # 40        } inserted into program at
 STA 1024,X      } this point. It is defined
 DEX             } here at the first occurrence.
 BNE 772         }

+END             } Signals the end of the macro
  .              } definition.
  .
+MACRO1            Assembler inserts a second
                   copy of the macro here.
  .
 RTS
 END
```

Fig 5.3

When using labels, label references and macros, it should be borne in mind that the assembler needs to store the names and locations during the assembly process. The version of assembler supplied on the disc will accept up to a maximum of 20 labels or memory labels, 20 label references and 10 macros. Macros which contain labels or references create new labels and references and these must also be taken into account. Thus, repeating a macro with 1 label and 2 references creates an extra 2 labels and references + one macro.

1. If the numbers of labels, references or macros likely to be used is going to exceed the numbers provided for in the program then it may be necessary to re-dimension the arrays holding the labels, label references or macros:

   a) Labels and memory labels are stored in F$(, F(which are dimensioned in line 5090 of the assembler program but to alter the number of labels that can be used, line 5080 should be altered i.e. to enable 30 labels to be used. This line should be altered to contain: 5080 D2=30.

   b) Label references are stored in K$(, K(and K1(which are dimensioned in line 5090 but the value used in dimensioning them is in line 5080. As the program is written, altering D2 also alters the dimensioning of the label referencing arrays.

   c) Macros are stored in M$(, M(and M1(which are dimensioned in line 5090 but the value is defined in line 5080. To enable 15 macros to be used, line 5080 should contain: 5080 D3 = 15

2. As you are aware, the APPLE is blessed with a large memory, and, as a consequence it is extremely unlikely that space problems are likely to occur with the assembler. However, for the record:

   a) Each occurrence of the label suite of variables F$(and F(requires 14 bytes.

   b) Each occurrence of the label reference suite K$(, K%(and K1%(requires 21 bytes.

   c) each occurrence of the macro suite M$(, M%(and M1%(requires 21 bytes.

A reasonable rule of thumb is that each new label, reference or macro requires about 20 more bytes.

Further Options of the Assembler

One of the options which is offered on the main MENU is 'Other functions - O'. If this is selected, the screen will display an alternative MENU which provides you with additional functions for inserting or moving code (I), or accessing the Monitor (M), or converting a machine code program to BASIC DATA statements (D), or obtaining an assembly listing on the Printer, instead of the screen (P), or loading program (L) or saving a program, or, finally, the option to return to the main MENU (X). The 'Monitor - M' option will be dealt with in the next chapter, the other functions are described below.

Inserting or Moving Code

The INSERT function is accessed by selecting an 'O' at the main MENU, an 'I' at the second MENU, and responding 'I' when asked "Insert..I : Move..M".

This facility will be found to be of most use when you need to insert a new section of code into an existing program. The need for this can arise through design or accident (planned expansion, or missed lines). The INSERT facility allows you to open up a gap in the program, into which you can enter new code (using the E option on the main MENU).

In the example shown below, a six byte gap is opened up in an existing program which extends from 828 to 842.

i) Enter start address of code to be entered, i.e. the address of the start of the gap, when asked:

   START ADDRESS FOR INSERTION? 832

ii) Enter the address of the end of the current program when asked:

   END OF CURRENT PROGRAM? 842

iii) Enter length of the code to be inserted, in bytes, when asked:

   LENGTH OF INSERTION? 6

iv) When space has been created for the new code, the screen will say:

OK SPACE INSERTED

v) Press any key to return to the main MENU and then use the normal ENTER procedure to enter the insert by selecting 'E'.

To access the MOVE function, much the same path is followed as for the INSERT function, but selecting 'M' when asked "Insert..I : Move.. M".

In the example below, a machine code program originally written into 768 to 842 is to be moved to just below DOS.

i) Enter the start address of the program to be moved when asked:

OLD START ADDRESS OF BLOCK? 768

ii) Enter the end address of the program when asked:

OLD END ADDRESS OF BLOCK? 842

iii) Enter the address where the program is to start, after moving, when asked:

NEW START ADDRESS OF BLOCK
.....i.e. after move ? $9500

iv) When the block has been moved, the screen will display:

OK - BLOCK MOVED

v) Press any key to return to the main MENU.

Converting a Program to BASIC DATA Statements

One way of entering a machine code program from a BASIC program is to convert the program into a series of numbers in DATA statements. The BASIC program then has to POKE the numbers back into the appropriate locations using a simple loop.

The data statements generated by the program are written to disk as a text file so they can be EXECed in to the program you wish them to be included in.

In the example below, a machine code program starting at 768 and finishing at 787 is to be converted into BASIC DATA statements.

'O' has been selected at the MAIN MENU and 'D' at the OTHER OPTIONS MENU.

i) Enter a line number between 0 and 63999, preferably where you want the section of code to start in your BASIC program, when asked:

LINE NUMBER FOR 1ST DATA STATEMENT? 20000

The number is a reasonably high number so that it is not likely to interfere with any program it might be used in.

ii) Enter the start address of the program which is to be converted to DATA statements when asked:

START ADDRESS OF DATA? 768

iii) Enter the address of the end of the program when asked:

END ADDRESS OF DATA? 787

iv) Enter the name of the file name you wish the DATA statements to be saved in when asked:

WHAT FILE NAME DO YOU WISH TO USE? PROGRAM

The data statements will then be saved in a file called PROGRAM.DATA.

v) The disk drive will start up and the DATA statements will be printed on the screen and to the file. After printing the DATA statements, the screen will then display the message:

DATA STATEMENTS NOW ENTERED

Pressing any key will return you to the MAIN MENU.

Printing the Assembly Listing on a Printer

The 'L' option on the main MENU allows you to display the assembly listing on the screen. If you own a printer however, you will want to print your listings. The 'P' option on the OTHER OPTIONS MENU provides this facility. The printed output will be exactly the same as the screen listing.

At the MAIN MENU select 'O', then select 'P' at the OTHER OPTIONS MENU. Make sure that your printer is connected and switched on, of course. Now provide the start address of the program and the end address when asked to do so, and the printer will produce your listings, followed by the message:

OK-Program Listed

and you can press any key to return to the main MENU.

Saving a Program

This option allows you to save your machine code program to disk either for future use in your own program or for reloading to save you the trouble of re-typing it in.

It is accessed by typing "S" from the OTHER OPTIONS MENU.

If you wish to save a program called MULTIPLY which starts at 768 and ends at 800 you are first asked:

```
WHAT IS THE NAME OF THE PROGRAM YOU
WISH TO SAVE? MULTIPLY
```

You are then asked where the program starts.

```
ENTER START ADDRESS? 768
```

And where the program ends.

```
ENTER END ADDRESS? 800
```

The program will then be saved under the name MULTIPLY and you will then be returned to OTHER OPTIONS MENU.

TYPING "?" in response to the prompt for the program's name would have printed the catalogue of the disk in the disk drive.

Loading a Program

It is accessed by typing "L" from the OTHER OPTIONS MENU. If you wish to load a program called MULTIPLY you are then asked:

```
WHAT IS THE NAME OF THE FILE YOU WISH
TO LOAD? MULTIPLY
```

The program is then loaded into memory and you are returned to the OTHER OPTIONS MENU.

Typing "?" in response to the prompt for the program's name would have printed the catalogue of the disk in the disk drive.

- - - - -

CHAPTER 6

Without the Assembler

So far all the machine code programs considered have been entered via the assembler. However, as discussed in the last chapter, all the assembler does is to make it easy to POKE data into memory. The assembler is, therefore, not the only way of putting a machine code program into memory: we can POKE it in directly. Program 6.1 is expressed in this direct POKE form.

PROGRAM 6.1

```
POKE 768,162
POKE 769,0
POKE 770,169
POKE 771,90
POKE 772,157
POKE 773,0
POKE 774,4
POKE 775,232
POKE 776,208
POKE 777,250
POKE 778,96
```

We cannot enter this into memory, of course, while we are running the assembler. So first of all, select the X option on the main menu to exit and confirm your choice by typing Y. The computer will respond by clearing the screen and displaying the BASIC prompt. When the program is in, it may be run by the CALL 768 command and it will print 256 flashing 'Z's, although you will not be able to see them all. Can you imagine the program that these POKEs have created?

Fortunately, it's not necessary to imagine it, as the assembler will disassemble the code from memory for you and display it. First RUN the assembler by typing RUN, then select 'L' at the MENU and type in the address 768.

The assembler will then reveal the source program, which was:

PROGRAM 6.1a (In Assembly Language)

```
LDX #0
LDA #90
STA 1024,X
INX
BNE 772
RTS
```

As well as entering programs without the assembler it is also possible to run them directly, or from a BASIC program. For instance, to run a program which starts at memory location 768 it is only necessary to put a 768 into the program counter. If Program 6.1 is still in memory this can be used to demonstrate this - if it is not still in memory type it in. Next exit from the assembler and get back to BASIC as before.

Now type in CALL 768 and the program will run, putting in the 'Z's.

Now to run it from BASIC. Put in the following program:

PROGRAM 6.2

```
20000 HOME
20010 CALL 768          Direct PC to 768
20020 PRINT "ALL OVER"
20030 END
```

Now type in "RUN 20000" <return> and the BASIC program should run in the following way:-

Line 20000 The BASIC program clears the screen..
Line 20010 Hands control to the machine code program at 768. Control is handed back to BASIC when the final RTS is encountered.
Line 20020 BASIC prints out the message "ALL OVER".

Running a program directly is relatively easy but the direct mode of entry of a program is obviously a tedious way of entering a long program so a further, more easily entered option is offered by the storing of the machine code in DATA statements. Program 6.3 shows a machine code program loaded via BASIC:

PROGRAM 6.3

```
1 FOR X = 0 TO 10
2 READ A
3 POKE (768 + X),A
4 NEXT X
5 DATA 162,0,169,90,157,0,4,232,
208,250,96
6 END
```

This is run as for a normal BASIC program with a RUN command. Once run, the DATA will have been loaded into memory and lines 1 to 7 must be looped out.

Once this is done, the main assembler can be run and the program RUN from the MENU - it's located at 768. Of course, once the data is loaded into memory, the program can also be run by a CALL 768 command.

The Dr Watson Assembler for the APPLE offers a further method of entering machine code via:

The Monitor: Option M (of 'other functions')

This feature offers a ready feature for examining and modifying memory. To enter the monitor, type "M" at the OTHER OPTIONS MENU. The computer will then clear the screen and the prompt character will change from Applesoft's ']' to the monitor's '*'.

The monitor accepts only single letter commands optionally preceded by an address. To investigate this type in the command:

```
*300L
```

This activates the mini-disassembler to disassemble 20 instructions starting at $0300. If Program 6.1 is still in memory, the display will look something like this:

```
0300-   A2 00        LDX   #$00
0302-   A9 5A        LDA   #$5A
0304    9D 00 04     STA   $0400,X
0307-   E8           INX
0308-   D0 FA        BNE   $0304
030A-   60           RTS
030B-   00           BRK
030C-   00           BRK
030D-   00           BRK
030E-   00           BRK
030F-   00           BRK
0310-   00           BRK
0311-   00           BRK
0312-   00           BRK
0313-   00           BRK
0314-   00           BRK
0315-   00           BRK
0316-   00           BRK
0317-   00           BRK
0318-   00           BRK
```

As you can see the display is similar to the assembler's list option but all addresses are in hexadecimal as the monitor will not accept decimal numbers, only hexadecimal numbers.

The monitor will allow you to alter memory in a limited way by changing individual memory locations by overwriting them. To demonstrate this, try typing in the following:

```
300:A0
303:20
304:99
307:C8
```

If you now list Program 6.1 (*300L) you will see that it looks like this:

```
0300-   A2 00        LDY   #$00
0302-   A9 20        LDA   #$20
0304    99 00 04     STA   $0400,Y
0307-   C8           INY
0308-   D0 FA        BNE   $0304
030A-   60           RTS
030B-   00           BRK

            ETC.
```

The program may be run from the assembler by typing CTRL Y to exit from the monitor and return to the OTHER FUNCTIONS MENU and then run it as normal.

The program can also be run by the monitor command "G", which is short for go! It runs a machine code program directly from memory. Naturally the Apple has to be told where the program starts and so the total command reads:

```
300G
```

Protecting Machine – Code in Memory

If you wish to use a machine-code routine when you are running a BASIC program then you need to place it somewhere in memory where it will not be overwritten by BASIC.

Page 3, from $0300 to $03CF (768 to 975) can be used for small programs and it is often used by other people's programs!

Programs larger than 208 bytes long can be stored below DOS' buffers if both DOS and BASIC know about it as BASIC stores its strings from HIMEM downwards in memory and DOS moves HIMEM up or down depending on how many buffers it has to maintain.

DOS

You can still use DOS from the monitor as if you were in BASIC. However, if an error occurs when using a DOS command, you will find yourself back in BASIC. To return to the monitor type:

```
CALL-151
```

and you will be back in.

If DOS commands do not work from the monitor, try typing:

```
3EAG
```

This should re-connect DOS and leave you still in the monitor.

SAVE

As you can use DOS from the monitor, you may wish to save a program to disk. To do this you need to know where the program starts and how long it is. In the case of a program that starts at $0300 and ends at $030A, it is #0B (11) bytes long. Now we have this information we can use the BSAVE command to save it.

The general format of the BSAVE command is:

```
BSAVE program name,A start address,L length
```

The start and length information may be in either decimal or hexadecimal, hexadecimal numbers being preceded by a dollar sign.

To save Program 6.1, the command is something like:

```
BSAVE PROGRAM 6.1,A$300,L$B
```

or

```
BSAVE PROGRAM 6.1,A768,L11
```

LOAD

To load the program back in at the same location, it is only necessary to type:

```
BLOAD PROGRAM 6.1
```

as the program's start address and length are saved with the program. If you wish the program to start at a different location from the one it was saved at, for example $030B, the command is changed to:

```
BLOAD PROGRAM 6.1,A$30B
```

Register Display and Debugging

When using the G command to run a program, the 6502's registers can be set so you know exactly what they are. To examine the registers type:

```
<crtl>E
```

The registers will then be printed out:

```
A=00 X=88 Y=DB P=00 S=10
```

They can be altered by using the colon command, ":" followed by the new values for the registers. If you wish to change the contents of a register other than the accumulator then all registers up to that point also have to be entered, for example to change the Y register to $00 while leaving the other registers with their present values you would have to type:

```
:00 88 00
```

You can only alter the register contents if the previous line is a display of their current contents.

One of the 6502 instructions that has not yet been dealt with is BRK which has a hex value of 00. If the 6502 encounters a BRK instruction then the normal flow is 'interrupted' (a full description of the BRK command and of machine 'interrupts' is given in Chapter 8 but a brief treatment is given here). Normally on the Apple a BRK command will beep the speaker and leave you in the monitor although you can alter this.

Sometimes the reason for the BRK is simply that the program has gone beserk and is executing data as if they were instructions: there are always a large number of zeros hanging around in a computer's memory and these will be interpreted as BRK instructions. This is good since you will (a) have regained control of your runaway program and (b) know which part of memory it was running away in.

However, a program behaving like a rogue elephant is not the only reason for a break entry to monitor. A much better reason is that you yourself organised the BRK. Suppose that you have a machine code program which is not doing what you expect, and you are unable to determine what is going wrong in spite of bringing all your intellectual power to bear on the problem. Don't give up! There is a way forward.

You can insert a BRK instruction into your program and run it. When (if) the BRK instruction is executed you will arrive in monitor with values of all the registers shown to you. You can now use the L command to look at various instructions stored in those parts of memory which you have been messing about with and this, hopefully, may give you the clue that you need. If you don't arrive in the monitor by the way, that in itself is vital information, since presumably, you placed the BRK instruction in a part of the program that you expected to be executed so either the program can't get that far or it is taking an unplanned route.

You can seed your program with as many BRK instructions as you wish. There is no problem of identification as the monitor tells you the value of the program counter when it is entered, so you will be able to identify which of your many BRKs caused the entry.

When you try your first BRK program, you will discover a curious thing. The program counter which is shown is not the address of the BRK instruction but is two bytes further on.

Let us have a look at the display caused by a BRK at $0800.

```
0802-  A=99 X=88 Y=DB P=30 S=D1
```

This displays the address plus 2 of the BRK command and the contents of the registers for examination.

You may feel that putting extra BRK commands into your program is a bit of a nuisance, especially if your program is long. In this case, you could use the L and : commands to change the value of an existing instruction to 00, thereby changing it to a BRK command. If you do this, you must remember before re-entering the program to:

(a) Use a value of 2 less for the program counter than the value displayed by the break and

(b) Restore the value of the instruction you changed (the trick here is to make a note of the hex value BEFORE you change it!)

I am sure you can see what a powerful tool this gives you for debugging your machine code program. You will be surprised how useful this can be.

One facility of the monitor's you may find useful is its ability to add and subtract eight bit hexadecimal numbers. To add say $88 and $31 type:

```
88+31
```

and the computer will print:

```
=B9
```

but if you type:

```
FF+1
```

you will see:

```
=00
```

because the carry (or borrow in subtraction) is ignored.

Summary of Monitor Commands

<address>L	Disassemble 20 instructions
<address>:<byte1> <byte2>...	Change memory
<CTRL>E	Display registers
BSAVE<FILENAME>,A<start address>,L<length>	Save block of memory
BLOAD<FILENAME>	Load block of memory
<address>G	Run the program starting at <address>
<CTRL>Y	Return to assembler

This is not an exhaustive list of monitor functions but they are probably the most useful.

CHAPTER 7

Built-In Subroutines

As the APPLE itself uses the 6502 chip, it has stored inside it, in ROM (Read Only Memory), machine code routines that control the 6502. These enable the APPLE to deal with the BASIC commands that are put into programs, all input and output and all the standard routines which are needed to keep the APPLE alive and well. The ROM which handles all the BASIC statements is located in memory between $D000 and $F7FF. The ROM which looks after all the non-BASIC routines is called the Monitor and lies between $F800 and $FFFF.

In addition to this use of memory, both BASIC and the monitor make use of the RAM memory in the bottom four pages between $0000 and $3FFF, the most frequently used locations being in the zero page $00 to $FF. Some of this usage is for the storage of transient data such as, for instance, the random number seed ($4E,4F), some of RAM is used for more permanent data, such as the pointers stored in 103 to 116 ($67 to $74) which indicate the area of memory used by the BASIC program and its data areas. Some of these uses take up only one two or three bytes, others use much more, such as the input buffer which uses all of page 2.

The most difficult aspect in using the built-in subroutines is to know where they obtain their data from and where they deposit the data that they have generated. This is especially true of the routines which make up the BASIC ROM.

First let's look at the contents of the accumulator using an APPLE monitor subroutine. We have already displayed the accumulator by using a STA command to move a copy of the accumulator to a screen location, e.g. STA 1024. A better/easier method, however is to use the monitor subroutine which is called COUT and is located at 65005 ($FDED). This will output the accumulator to the screen starting from the current cursor position. This is illustrated in:

PROGRAM 7.1

```
LDA #170     Load accumulator with '170'
             (an asterisk).
STA 1468     Store accumulator in the middle
             of the screen.
JSR $FDED    Jump to COUT subroutine.
RTS
END
```

When run, this displays two asterisks. In the middle of the screen at 1468 an asterisk which we placed directly. However, we also have another asterisk, in the top left hand corner of the screen. This extra asterisk was placed there by the ROM subroutine. Notice that the subroutine didn't have to be told where the asterisk was to be placed, the subroutine placed the asterisk in the current cursor position.

The major advantage of this subroutine is that it locates the cursor automatically and will increment this automatically each time the subroutine is called. If you ran the program from the assembler then the current cursor position would have been 1024, because the assembler does a 'clear screen'.

If we wish to, we can set the current cursor position from our machine code program. This could be quite complicated, but for the presence of another built-in subroutine called TABV which is located at 64347 ($FB5B). This does most of the hard work for us in calculating the address of the start of a line, it is then only necessary to set up the cursor's position in the line.

To move the cursor to the start of a line, it is only necessary to load the accumulator with the line number and call TABV and to save the position of the cursor in the line in 36 ($24).

Let us use the TABV subroutine to place an asterisk at the beginning of the tenth line on the screen.

PROGRAM 7.2

```
LDA #9       Load 9 for tenth line
JSR $FB5B    Call TABV to position cursor
LDA #0       Load 0 for first position in line
STA 36       Address of offset along line
LDA #170     Load asterisk
JSR $FDED    Jump to COUT subroutine
RTS
```

If we had wished to put the asterisk at the eighteenth position of the tenth line then we would have saved 17 in 36.

PROGRAM 7.2a

```
773 LDA #17
    STA 36
```

PROGRAM 7.3

```
    LDA #9
    JSR $FB5B
    LDA #17
    STA 36
    LDA #170
    JSR $FDED
    RTS
```

When run, this program moves the cursor down 10 and across 18, and prints an asterisk at this location.

To illustrate how the COUT routine updates the cursor so that it will move to the next cursor position following each call, modify Program 7.3 by:

PROGRAM 7.3a

```
779     LDY #4
*LOOP   JSR $FDED
        DEY
        BNE 781
        RTS
        END
```

To yield:

PROGRAM 7.3b

```
        LDA #9
        JSR $FB5B
        LDA #17
        STA 36
        LDA #170
        LDY #4
*LOOP   JSR $FDED
        DEY
        BNE 781
        RTS
        END
```

When run, Program 7.3b will print four asterisks in line 10 in columns 18, 19, 20 and 21. Notice that we didn't have to reload the accumulator with 170 each time round the loop, so the COUT subroutine did not alter the value of the accumulator. It is also clear that the Y register is not altered, otherwise the counting of the four asterisks would not have worked. In fact COUT doesn't alter either A, X or Y. This is one of the good features of this particular routine. Not all built-in subroutines are so kind, so it is important to bear in mind the possibility of one or more of the registers being altered by any subroutine that we choose to use from the monitor or BASIC ROMs.

Many BASIC programs use the GET command, which accepts a single byte from the keyboard. GET uses one of the monitor subroutines to carry out this operation called RDKEY at 64780 ($FD0C). When called, RDKEY waits until a key is pressed and returns it as an ASCII value in the accumulator. Program 7.4 shows RDKEY in operation.

PROGRAM 7.4

```
JSR $FD0C
JSR $FDED
RTS
```

When run, this program waits for a key to be pressed then prints it on the screen.

GETLN at 64874 ($FD6A) is an alternative input monitor subroutine to RDKEY. When inputting from keyboard, its action is similar to the BASIC INPUT statement i.e. the first time that GETLN is called, the characters that are typed in are stored in the keyboard buffer which starts at 512 ($0200), any editing which is done during the typing such as using ESC I, J, K, M will be applied. However, we do not need to organise the retrieval of the characters from this buffer, as the characters will be stored in sequence in the buffer. There is no need to organise the display of characters as this also is organised by GETLN. This produces the shortest program yet in 7.5!

PROGRAM 7.5

```
JSR $FD6A
RTS
```

The RDKEY subroutine can be used to design your own INPUT routine. For instance, you could use RDKEY to enter one character at a time checking, say, for a certain terminating character which need not necessarily be a RETURN. You could also set the routine to check each character in the actual INPUT itself and give a warning if it is an invalid character.

Program 7.6 shows an arrangement with a check built in to look for a comma (ASCII 172) to be input.

PROGRAM 7.6

```
            LDA #172    }Store terminator in 900
            STA 900     }
*LOOP       JSR $FDOC   }Get a character
            JSR $FDED   }Print it
            CMP 900       Look for comma, if not present
                          branch back.
            BNE LOOP    }
            LDA #141    }Output a <return> to screen
                        } to be tidy.
            JSR $FDED   }
            RTS
```

This program simulates an INPUT routine that is terminated by a comma instead of a RETURN. To use a terminator other than comma, change the operand of the first instruction so that 900 will be loaded with the correct value. Try this for:

> EXERCISE 7.1
>
> Modify Program 7.6 to accept an input that is terminated by a space, use a POKE command to make the change.
>
> Answer in Chapter 9.

As was mentioned earlier, one of the major problems encountered when using built-in subroutines is that they too use the 6502. That means that they put things into A, X and Y and they also modify the SR - the Z, N and other flags. Hence when returning from any JSR it's not reasonable to assume that everything is just as it was left before the JSR, unless, of course we know (as with the COUT routine) what is left undisturbed.

As an example of this, look at the following program that is designed to input a four-character string from the keyboard. Firstly it sets a loop counter in Y at 4, then it uses the RDKEY and COUT subroutines. On return from these it decrements Y and checks for Z flag set - all very straightforward!

PROGRAM 7.8

```
            LDY #4
*LOOP       JSR $FDOC
            JSR $FDED
            DEY
            BNE LOOP
            RTS
```

However, when run, this program RTS's after only two characters have been input! This suggests that one of the subroutines is using the Y register. To verify this the program can be made to print out the Y register at various stages just to check. This is done in Program 7.9, where the contents of the Y register are examined immediately after returning from the subroutine.

PROGRAM 7.9

```
            LDY #4
*LOOP       JSR $FDOC
            STY 1024
            JSR $FDED
            STY 1026
            DEY
            BNE LOOP
            RTS
```

When run, this program will print the first character you type at the top left of the screen, overprinting the first value of Y and printing an inverse '@' two positions along for the second value of Y in the loop. The second time you press a key, first an inverse 'A' is printed followed by the key you had just pressed followed by another inverse 'A' at which point the program returns you to the assembler prematurely. This is because the Y register holds the value of column the second key was printed in, which would have been 1 instead of 3.

To overcome this problem, the Y register value must be stored somewhere prior to entering the subroutine and then retrieved prior to being decremented. Program 7.10 shows this process, where Y is temporarily stored in 900 during the subroutine.

PROGRAM 7.10

```
                                        Assembled Version
@LOOPCONST 900
           LDY #4                       LDY #4
*STORE     STY LOOPCONST                STY 900
           JSR $FDOC                    JSR 64780
           JSR $FDED                    JSR 65005
           LDY LOOPCONST                LDY 900
           DEY                          DEY
           BNE STORE                    BNE 770
           RTS                          RTS
```

When run, this program allows four entries from the keyboard, displays these and then passes out of the loop.

As a technique, the use of memory storage in this way does work, but it does call for some care in storing the data safely and retrieving it when needed. Fortunately the 6502 has a device for doing this operation automatically. It is

The Stack

The stack (S) is a block of memory, on the APPLE machines located from 511 ($01FF) down to 256 ($0100) capable of holding 255 bytes. It is used for the rapid transfer of data and is filled downwards from 511, its next vacant location being recorded by a STACK POINTER (or SP as it is known; this was shown as the last value in the line which was displayed by Monitor in response to the <CTRL> E command). When anything is to be retrieved from the STACK only the last item put in is accessible. The usual analogy is with a stack of plates, only the top one being accessible as this was the last one put there. However, the 6502 stack is filled DOWNWARDS, i.e. from 511 towards 256, so plates are put at the bottom and retrieved from there, antipodean fashion! This mode of filling and emptying the stack is known as last-in, first-out, i.e. the stack is a last-in, first-out (LIFO) store.

One function of the stack is to record addresses during subroutine jumps, which it does automatically. When the 6502 sees an instruction such as JSR 50000 it must first of all record where the next instruction is so that it can find its new location after the subroutine has been executed and then place the "50000" into the program counter (PC). The process is examined below with the program segment 7.11 (from 7.10).

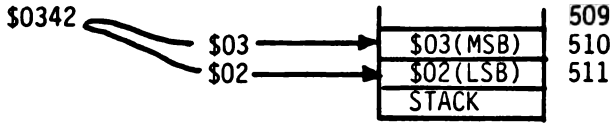
PROGRAM 7.11

```
STEP 1.  768 ($0300)  STY 900    ($384)
STEP 2.  770 ($0302)  JSR $FDOC  (64780)
STEP 3.  773 ($0305)  JSR $FDED
```

INSTRUCTION STY 900

STEP 1.
- i) Calculate address of next instruction, i.e. 770 or $0302.
- ii) Put next address into program counter.
- iii) Execute instruction STY 900.
- iv) Retrieve address for next instruction from PC, i.e. $0302.

STEP 2.
- v) Fetch next instruction, i.e. at $0302. This is JSR $FDOC.
- vi) Retrieve address for next instruction in program, i.e. $0305, and place this on the stack.



- vii) Record the next vacant location in the stack pointer, i.e. 509 (SP=509).
- viii) Load $FDOC into program counter.
- ix) Jump to subroutine at $FDOC.
- x) Execute subroutine until RTS is encountered.
- xi) Look at stack pointer to find last data. SP=509, therefore Data stored at 510 and 511.
- xii) Extract data from 510 and 511 (i.e. $0305) and load this into PC, reset SP to 511.
- xiii) Jump to $0305.

STEP 3.
- xiv) Fetch next instruction and carry on with program.

In this example, the first subroutine could have met nested subroutines, and each time a JSR was executed the return address would have been piled on top. Then, as the program returned successively through these subroutines, the return addresses would have been stripped off to steer the 6502 back to the original point of departure. Fig. 7.2 illustrates this for subroutines nested 3 deep starting from a program with a JSR $0384 instruction in $033D.

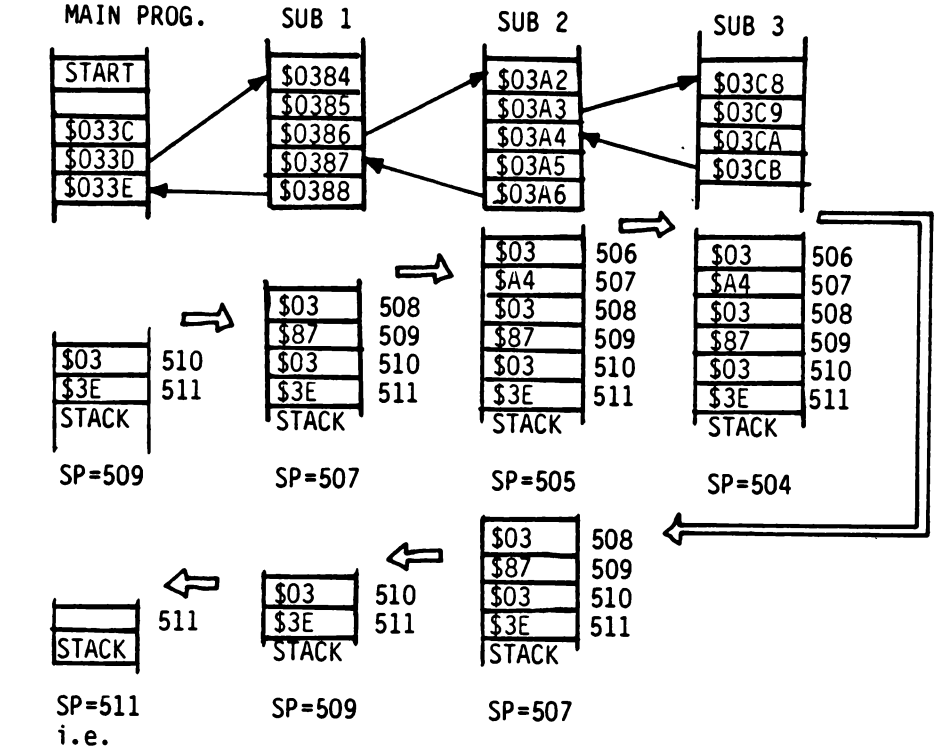


Fig. 7.2

In the above program, the instructions at $03CB, $03A6 and $0388 would all be RTS. Thus when the 6502 finds an RTS at $03EB, it takes the top address off the stack, which is $03A4. Reading downwards at SUB 3, the stack gives

i) address for return to subroutine 2.
ii) address for return to subroutine 1.
iii) address for return to the main program.

Fortunately, the operation of the stack in recording addresses when executing subroutines is automatic and so the programmer can allow the 6502 to do the job. However, as was seen when using built-in subroutines, the stack does not automatically store register contents but must be programmed to do so. Only two instructions exist for storing registers, neither of these operating on the X and Y registers. These must be handled via the accumulator which is stored using:-

PHA	PusH contents of Accumulator onto stack.

The contents may then be retrieved by means of:-

PLA	PuLl top of stack into Accumulator.

In both cases, the stack pointer is updated appropriately so that it continues to point to the next empty location in the stack.

Using these instructions, program 7.8 can be rewritten to transfer the Y register into the stack and retrieve this when needed. The stack pointer will take care of the order of the data, providing that the LIFO structure of the stack is borne in mind and data is entered and retrieved in the right order.

Writing these in yields:

PROGRAM 7.12

```
LDY #4
TYA  }     Set up loop constant.
PHA  }     Transfer Y into stack.
JSR $FDOC
JSR $FDED
PLA  }     Recover Y from stack,
TAY  }     decrement and check
DEY  }     for end of loop.
BNE 770
RTS
```

When run, the program accepts a four character input and prints this on the screen.

As well as affecting the accumulator, a subroutine is most likely to reset one or more flags in the SR. On returning to the main program, the reset flags are then certain to upset the course of this. To overcome this problem, the 6502 has built-in provision for saving the SR (i.e. the condition of all the flags on the stack). This is brought about by the instruction:-

PHP	PusH Processor status register on stack.

Retrieval of the data is achieved by:-

PLP	PuLl stack to Processor status register.

In Program 7.12 the SR was not saved on the stack as, prior to testing the Z flag with a BNE, the DEX instruction reset this. However, under other circumstances it may have been necessary to preserve the state of the SR so this should be written into 7.12 as an exercise.

EXERCISE 7.2

Rewrite Program 7.12 so as to save the condition of SR in the stack prior to the subroutines and retrieve this after these.

A possible answer in Chapter 9.

When using the stack, the main precaution to take is to check the order of entry and retrieval several times - always LIFO. For instance, a possible routine for saving the accumulator, X and Y registers and the SR is given in Fig. 7.3.

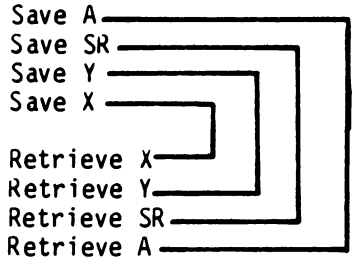


Fig. 7.3

CHAPTER 8

Interrupting the 6502, Variables

WHEN DOING A JOB THAT HAS TO BE DONE, NO ONE LIKES interruptions until the job's finished. The 6502 is just like that too! During a piece of program it has its Accumulator and X and Y registers under control and all the flags set appropriately. Thus, when an interrupt comes all these registers have to be stored, usually in the stack, and after the interrupt they must all be retrieved. An interrupt is, in fact, only a subroutine that comes when IT is ready and not the 6502! This means that the interrupt is generated from outside the 6502's cosy system, from an external device.

The handling of an interrupt has to be prepared for in the program and at certain times the program may well not be able to allow an interruption. If, for instance, another device is sending a stream of data into memory, then a HAND-SHAKING procedure is carried out between the two machines. Quite simply, this is an exchange of messages along the line: "I am ready to send data, are you ready to recieve it?" "yes." "Here's the data .....end of data." "Thanks!". If such an exchange is interrupted, then the data is likely to become garbled, and hence worthless. During such periods when no interrupt is allowable, the program can block most interrupts - not all - to allow a particular process to be completed. The instruction that allows this blocking is:-

SEI SEt Interrupt Disable Flag (prevent interrupts).

Ironically, the first action that usually needs to be taken in an interrupt is to set the I (interrupt disable) flag use of SEI. We need to prevent any more interruptions, at least for a while, so that we can check up to see if the interrupt which has occurred is our interrupt (there may be other potential interrupts lurking around). When we have made sure that the interrupt which has taken place is the one we are interested in, then the interrupt disable flag (or simply I flag) may be reset to '0' or cleared by the instruction:-

CLI CLear Interrupt Disable Flag (allow interrupts).

If you think about the above, you will realise that we may allow interrupts to be interrupted. That is perfectly true. It is rather like someone starting to peel the potatoes when the kettle interrupts by coming to the boil, so the potato cleaning task is suspended for a while and a start is made on making a pot of tea. At this point the milkman, who wants his milk bill to be settled, interrupts by ringing the doorbell. So the tea making task is suspended and the milkman is attended to. In the middle of this, the telephone rings ... After dealing with the telephone call, then the milkman paying task may be completed, then we must finish making the pot of tea, at which time we can go back to peeling the potatoes. The trick is to remember how far you have got with each task when it is restarted. Unlike human beings, the 6502 is very good at remembering where it has got to, and has little difficulty.

Not all interrupts can be blocked by setting the I (interrupt disable) flag, as some are crucial and must get through at all costs. Many such circumstances may arise during the control of plant or when a power failure calls for immediate action. To enable the 6502 to distinguish between the two conditions, it has two different input pins, one for each type of interrupt. One of these is the NMI or Non Maskable Input pin which cannot be blocked, and the other the IRQ or Interrupt ReQuest pin which is masked off by the I flag.

When the 6502 receives an interrupt signal it always completes its current instruction before doing anything else. In the case of an IRQ interrupt it would then check the I flag and if not clear, continue until the program cleared this. Next, before going into the interrupt procedure, the contents of PC are saved on the stack (telling the 6502 where to return to when it has finished processing the interrupt), and then the SR (status word) is saved. This saving of data is really only a half-measure as, almost certainly, A, X and Y would be changed during the interrupt precedure.

Next the I flag is set to 'I' to prevent further interrupts and then the appropriate address for the appropriate interrupt routine is loaded into the PC. These addresses are found at 65530 and 65531 ($FFFA and $FFFB) for an NMI interrupt and 65534 and 65535 ($FFFE and $FFFF) for an IRQ interrupt. These interrupt routines are terminated with an instruction:-

RTI ReTurn from Interrupt.

On meeting this, the 6502 does three things:
(a) restores the SR so that the status flags are the same as when the interrupt occurred,
(b) resets the I flag - an automatic CLI, and
(c) looks in the stack for the return address and stores it in the SR - like an automatic RTS.

Unfortunately, the 6502 only does half the job at an interrupt and, as discussed on pages 7-9 and 7-10, A, X and Y should be saved on the stack. In the case of both X and Y this must be transferred into A before being pushed onto the stack (by a PHA), and when pulled off the stack (by a PLA) it will need to be restored to the appropriate register by a TAX or TAY.

The preceding section was included for completeness as the standard Apple does not generate interrupts. However, certain peripheral cards may generate interrupts such as clock cards.

The 6502 possesses one other interrupt instruction:-

BRK BReaK

When the BRK instruction is encountered, the 6502 first resets the PC by indexing this by one place (so that the PC points to the byte following the BRK instruction), and stores this address on the stack then it sets the Break Flag (B flag) which is bit 4 of the SR and stores it on the stack also. Following this, the 6502 then does a normal IRQ interrupt using the IRQ VECTOR at $FFFE (LSB) and $FFFF (MSB). The IRQ routine will check whether this interrupt was caused by a true IRQ or a BRK instruction.

Normally, on the Apple, the IRQ routine will jump into the monitor, unless the BRK vector at 1022 and 1023 ($03FE and $03FF) has been altered, and the prompt character is changed from a right square bracket, ']' to an asterisk, '*', great stuff for debugging!

To test this, run the following:-

PROGRAM 8.1

```
        LDA #218
        STA 1024
        BRK
```

When run, this will print a 'Z' in 1024 and then go into the interrupt routine and enter the monitor, beeping the speaker in the process to tell you that something unusual has happened.

By using the assembler to disassemble the code at $FFFE and $FFFF, you will be able to find this address and from there to follow thorugh the rest of this routine. By following this code through you will discover how the assembler traps the BRK command and enters the monitor. The vital element is the JMP IA(1008) instruction which you will find at 64086($FA56). 790 and 791 normally contain the address of the Monitor Break entry point.

When the 6502 chip is used as an element in a system, it is then that BRK comes comes into its own as the interrupt vectors, as they are called (addresses pointing to routines) at $FFFA and $FFFE can point to whatever routine the designers wish. On the later versions of the Apple II+ and IIe, the designers made the decision to point the 6502 along the path that will do least harm. However, by routing this routine through a vector in RAM which is accessible and changeable by the user, they give us the option of doing something different from running the monitor.

Signed Numbers

In all the mathematical exercises done so far, the numbers used have been treated as simple positive numbers. Thus, any arithmetical processes have dealt with these numbers as strings of eight bits. However, if negative integers are to be used in arithmetic, one of the bits must be used to indicate that a given byte represents a negative number. Bit 7, the left-most bit, is used to do this, being set to a zero if the number is positive and a one if it is negative. By using one bit in this way, the magnitude of the number stored in the remaining seven bits is restricted to +127 to -128. Conventionally a zero is used on bit 7 to indicate the presence of a positive number and a 1 for a minus number. One of the problems that arises from this usage is that, in theory, two forms can exist for the number zero, i.e. +zero and -zero:

+0 = 00000000

-0 = 10000000

In order to overcome the problem, the negative number is normally represented in what appears to be a weird form - TWO'S COMPLEMENT! Weird it may be, but it works!! In order to work out a two's complement representation, take the number 38_{10} which in binary is 00100110. To convert it to its two's complement negative form, first of all switch each bit of the positive number from a 0 to 1 or vice versa, i.e. to its COMPLEMENT:

$$00100110 \longrightarrow 11011001$$

Next one is added to this switched form or ONE'S COMPLEMENT, i.e.

$$\begin{array}{r} 11011001 \\ +\quad\quad 1 \\ \hline 11011010 \end{array}$$

This yields the negative two's complement representation, i.e.

$$-38_{10} = 11011010_2$$

To understand the significance of this representation, three sums using it are illustrated below.

i) 38 - 38, which should of course yield zero.

$$\begin{array}{r} -38 = 11011010 \\ +38 = 00100110 \\ \hline 00 = 00000000 \end{array}$$

ii) $43_{10} - 3810$

$$43_{10} = 00101011_2$$

$$-38_{10} = 11011010_2$$

i.e. $-38 = 11011010$

$$+43 = \underline{00101011}$$

$$= \underline{00000101 = 5_{10}}$$

iii) $24_{10} - 3810$

$$24_{10} = 00011000_2$$

i.e. $24 = 00011000$

$$-38 = \underline{11011010}$$

$$\underline{11110010}$$

As this answer has a 1 in bit 7, it is a negative two's complement representation. To convert this, first find the one's complement:

11110010 00001101

Next add on 1

i.e. $\begin{array}{r} 00001101 \\ +\quad\quad 1 \\ \hline 00001110 \end{array} = -14_{10}$

Overflows

In signed number arithmetic the seven "magnitude" bits (i.e. 0 to 6) can only store a number up to +127, so any attempt to store a number greater than this will result in a 'carry' into bit 7, or as it is known in this case, an OVERFLOW. Consider the sum $100_{10}+30_{10}$.

i.e.
$$\begin{array}{rcr} 100_{10} & = & 01100100_2 \\ +\ 30_{10} & & +00011110_2 \\ \hline 130_{10} & & 10000010_2 \end{array}$$

Thus 10000010 is a negative number, as signified by the '1' in bit 7, and is therefore in two's complement form. To convert this, first find the one's complement:-

$$10000010_2 \longrightarrow 01111101_2$$

and then add 1

$$01111101_2 + 1 = 01111110_2 = 130_{10}$$

The 6502 handles this situation by monitoring the accumulator and, when an overflow occurs, setting the overflow or V-flag. This flag can be tested by the instructions:-

BVC	Branch on oVerflow Clear.
BVS	Branch on oVerflow Set

BVC tests the overflow flag and if it is clear, or not set (V=0), then a branch is executed.

BVS tests the overflow flag and if it is set (V=1), then a branch is executed.

When carrying out multiple precision arithmetic processes with signed integers,bit 7 must be treated as an internal carry and when an overflow occurs this must be transferred to the most significant byte.

Program 8.2 illustrates the use of BVC to test for an overflow as the accumulator is indexed with ones.

PROGRAM 8.2

```
CLC
LDA #0
ADC #1
BVC 771
STA 1024
RTS
```

When run, the content of the accumulator is increased progressively until the right-most seven bits are filled with ones. At the next increment, the seven ones reset to zeros and a carry is generated which pops into bit seven. This sets the carry bit and stops the branching, allowing the program to run through to the RTS. The STA 1024 then prints the value at overflow as an @ (i.e.128_{10}).

As with other flags, provision exists for the control of the overflow flag and it can be cleared by the instruction:-

CLV	CLear the oVerflow flag.

Unlike the other flags, however, the overflow flag cannot be set (as on the carry flag). It isn't really something that a programmer wants to do, in the normal way of things anyway, so the 6502 designers left it out of the instruction repertoire.

Numerical Screen Output

In all the numerical examples to date, screen output has been in APPLE display code. While it is possible to interpret this using a table, it is obviously necessary in a program to display numbers as numbers to the base 10. The major complication in this procedure lies in the fact that APPLE display code is effectively a base 256 representation and can thus display 0_{10} to 255_{10} using only one character, where base 10 displays would require up to three characters to represent the same value.

Program 8.3 tackles this conversion task and first sets out to find if the number is greater than 200 - i.e. first digit is '2' - or if it is less than 200 and greater than 100 - i.e. first digit '1'. It then checks in a similar way for the tens and units and uses the stack to store the remainder (i.e. so far unprocessed portion) of the number while adding the conversion constant (48) to the accumulator to change the binary value to the display value equivalent. In the example given, the number to be displayed - 152 - is loaded into the accumulator at the start of the program.

PROGRAM 8.3

```
            LDA #152        Put in number to be O/P.
            CMP #200        Compare A with 200.
            BCC ONEHUND     Branch if number is less than 200
            SBC #200        Remove left most digit.
                            (carry already set)
            PHA             Store remainder on stack.
            LDA #178        Load A with '2' for 2 * 100.
            STA 1024        Print A on screen.
            PLA             Retrieve A from stack.
            JMP TENS        Jump to tens routine.
*ONEHUND    CLC             Clear Carry.
            CMP #100        Compare A with 100.
            BCC TENS        Branch if less than 100.
            SBC #100        Remove left most digit.
                            (Carry already set).
            PHA             Store remainder on stack.
            LDA #177        Load A with '1' for 1 * 100.
            STA 1024        Print A on screen.
            PLA             Retrieve A from stack.
*TENS       CLC             Clear Carry.
            LDY #0          Set Y to zero.
            CMP #9          Compare A with 9.
            BCC ZEROTENS    Branch if A less than 9.
*LOOP       INY             Increment Y.
            SBC #10         Subtract 10 from A
                            (carry already set)
            CMP #9          Compare A with 9.
            BCS LOOP        Branch if A greater than 9.
*ZEROTENS   PHA             Store A on stack.
            TYA             Transfer Y to A.
            ADC #176        Add conversion constant  to A
                            (carry already clear)
            STA 1025        Print A on screen.
            PLA             Retrieve A from stack.
            ADC #176        Add conversion constant to A.
            STA 1026        Print A on screen.
            RTS
```

When assembled the program looks as in program 8.3a

PROGRAM 8.3a

```
LDA #152      } Check if number less than 200,
CMP #200      } If so branch.
BCC 786       }

SBC #200      }
PHA           }
LDA #178      } Subtract 200 from number and
STA 1024      } Print out digit '2' for 200.
PLA           }
JMP 800       }

CLC           } Check if number less than 100,
CMP #100      } if so branch.
BCC 800       }

SBC #100      }
PHA           } Subtract 100 from number and
LDA #177      } Print out '1' for 100.
STA 1024      }
PLA           }

CLC           }
LDY #0        } Check if remainder of number less
CMP #9        } Than or equal to 9; if so, branch.
BCC 814       }

INY           }
SBC #10       }
CMP #9        }
BCS 807       } Count number of 10s left in
PHA           } Remainder and print out 10s
TYA           } digit
ADC #176      }
STA 1025      }
PLA           }

ADC #176      } Print out unit's digit.
STA 1026      }
RTS           }
```

When run, this program will print '152' at the top left-hand corner of the screen. In general terms it can be used as a subroutine which prints out the contents of the accumulator.

Floating – Point Numbers

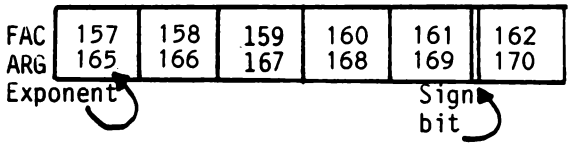
With the integers used so far, the scope of the arithmetical processes carried out has been somewhat restricted. When working in BASIC, however, the binary floating-point constants have 10 digit precision and are displayed with 9 digits. Their exponents have a range of -38 to +37. Each value is stored in six consecutive bytes of memory and, to ease the manipulation of these, two 'accumulators' are provided in memory locations $9D_{16}$ to $A2_{16}$ (157_{10} to 102_{10}) and $A5_{16}$ to AA_{16} (165_{10} to 170_{10}). These are known as the Floating-point Accumulator (FAC) and ARGUMENT REGISTER (ARG).

To store a number which could contain up to 10 digits when displayed in base 10 form, the floating point accumulators use only six bytes. How is this trick managed?

When running BASIC programs, you will have seen that very large or very small numbers are expressed in exponential form or scientific notation. Thus, 4079.013 can be expressed as 0.4079013E+4 or 0.4079013×10^{-4}, while 0.0000417 can be expressed as 0.417E-4 or 0.417×10^{4}. This representation contains two parts: the first part (the 0.4079013 in the first example) is called the MANTISSA, and the second part (the +4 of the 10^{4}) is called the exponent. These two parts are stored in binary in the floating point accumulators in the following manner:

(a) BINARY MANTISSA is stored in the middle four bytes of FAC and ARG. The sign of the mantissa is stored in the sixth byte where a '1' on bit 1 signifies a negative mantissa and a '0' signifies a positive mantissa

(b) BINARY EXPONENT is stored in the first byte of FAC and ARG. As this has to store both positive and negative exponents, the exponents needs to be converted to a positive form and this is done by adding 128. Thus, an exponent of 10 would be stored as 128+20, i.e. 48, while an exponent of -20 would yield 128-20, i.e. 108.



When loading a floating-point number, BASIC normalises its binary representation and thus its left-most (most significant) digit is always 1. Take, for instance, the number $+1400_{10}$ (0578_{16}). Expressed in binary this is $0000\ 0101\ 0111\ 1000_2$. In this form, the binary point (a binary number has a binary point instead of a decimal point to the right of the least-significant digit, i.e.

$0000\ 0101\ 0111\ 1000_2$.
Binary point

Putting the exponent in gives a representation:

$0000\ 0101\ 0111\ 1000_2 . *2^0$

and when NORMALISED, i.e. the binary point moved just past the left-most significant digit:

$.1010\ 1111\ 0000\ 0000 \quad *2^{11}$

i.e. the binary point has been moved 11 places to the left and so the number needs to be multiplied by 2^{11} to return to its original value.

Finally this is padded up with 0's in its least-significant bytes to form a four byte mantissa.

i.e. Byte 2 Byte 3 Byte 4 Byte 5

1010 1111 0000 0000 0000 0000 0000 0000

The exponent of 11_{10} is now converted to the correct format by adding 128_{10} to its raw form, i.e. it becomes 139_{10} or $1000\ 1011_2$ and this byte is added to the first location in the accumulator, i.e. the representation becomes:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
1000 1011	1010 1111	0000 0000	0000 0000	0000 0000

The sign of the mantissa is expressed in byte 6 by the condition of its most-significant bit, this being set to a '1' for a negative number and a '0' for a positive one. Thus, to represent $+1400_{10}$, bit 7 of byte 6 of FAC or ARG is set at '0'. The other bits of this byte may be set at some other number. Thus the final floating-point accumulator representation of $+1400_{10}$ is:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
1000 1011	1010 1111	0000 0000	0000 0000	0000 0000	0??? ????

To use this accumulator directly from machine-code is clearly none too easy, as the data has to be loaded into the memory locations in the correct format. However, some built-in subroutines are accessible to enable this to be done. For instance, a two byte integer can be converted into floating-point and loaded into the floating-point accumulator (FAC) by means of the subroutine at $E2F2(98098)

To accomplish this, the least-significant byte is loaded into the Y register and the MSB into the accumulator. Then a JSR $E2F2 will convert this integer into floating-point and load it into FAC. This subroutine is demonstrated below in program 8.4, where the LSB and MSB of an integer are converted into a floating-point number.

PROGRAM 8.4

```
LDY #LSB
LDA #MSB
JSR $E2F2
RTS
```

These two floating-point accumulators - FAC and ARG - are used for the manipulation of floating-point data and a means is provided for transferring floating-point data into them.

Taking ARG, for instance, the subroutine for loading floating-point data from memory resides at $E9E3 (59875) And it moves the contents pointed to by the Accumulator (LSB) and the Y register (MSB) into ARG. Thus, if the data is stored at $0350 onwards, program 8.5 carries out this transfer:

PROGRAM 8.5

```
LDA #80
LDY #3
JSR $E9E3
RTS
```

A similar subroutine re-locates data into FAC, this being located at $EAF9 (60153).

Unfortunately, as with the other routines using the floating-point accumulators, a measure of faith has been called for as in no case has the content of either FAC or ARG been visible. To rectify this a hybrid BASIC/machine -code command is utilised:

The USR Command

This command allows the transfer of data between FAC and a machine-code program. For instance, the line B=USR(Q) in a BASIC program causes the system to place the value of Q into the floating-point accumulator. It then jumps to a machine code routine whose address it finds at $0B (11) LSB and $0C (12) MSB. The presumtion is that you have placed a machine code routine into memory starting at that address and set the values of $0B and $0C to point to the routine. When your routine makes use of FAC it will, of course be using the value that was in Q when the USR function was called. When the particular arithmetic process has been carried out, your routine should leave the answer in FAC and, because of the assignment, this will then be placed into B by BASIC. As you would expect, we can use the USR function in the same way as any other function, for instance PRINT USR(P+2) will print out the result of the machine code routine which will have started with FAC containing the value stored in P incremented by 2.

To test this, a number can be placed into the floating-point accumulator by means of the USR function and then printed out. This is done by means of the two programs - program 8.6a and program 8.6b.

PROGRAM 8.6a

```
20000 HOME
20010 POKE 11,0        0=$00 } Address = $0300
20020 POKE 12,3        3=$03 }
20030 B=1400
20040 A=USR(B)
20050 PRINT"A=";A
20060 END
```

PROGRAM 8.6b

```
768 RTS
```

When this program is run, an address - $0300 (768) - is loaded into locations 11 and 12 by lines 20010 and 20020.

When line 20040 is run, the argument B (1400) is loaded into the floating-point accumulator. Control is then handed over to the machine code program at $0300 (768). At this point, the machine code routine doesn't actually modify the value of FAC, it simply obeys the RTS which will return control to BASIC. BASIC will place the contents of FAC into Q. Line 20050 prints out the value stored in Q, which has not been modified by the machine code routine.

This routine offers a way of loading any number, which is valid in BASIC, into the floating-point accumulator. It also offers a way - albeit rather round-about - of examining the contents of FAC. This is not as straight-forward as might be imagined, as most BASIC commands use the FAC when operating; thus even a PEEK command changes its contents. However, if the contents are examined in machine-code, immediately after being set they can be seen before BASIC gets its hands on them again. To do this, program 8.6b should be amended to examine memory locations $61 to $66 (97 to 102) and then to print these on the screen. This is done in program 8.7.

PROGRAM 8.7

```
LDX #6       Set up loop counter.
LDA 156,X    Load one byte of FAC.
STA 1152,X   Print on screen.
DEX          Decrement loop pointer.
BPL 770      Branch if more to do.
RTS
```

PRINT CONTENTS OF FAC ON SCREEN

As this program prints the contents in Apple display code, program 8.6a should also be modified to decode this - as below.

PROGRAM 8.8

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 B=1400
20040 A=USR (B)
20050 PRINT"A=";A
20060 PRINT
20070 FOR X=1 TO 6
20080 PRINT PEEK(1152 + X);" "
20090 NEXT X
20100 END
```

This program simply PEEKS the locations that display the contents and prints the answer.

A = 1400

and the contents:

139 175 0 0 0 47

When compared with the calculated contents on page 8-11 it will be seen that bytes 1-5 are identical, while byte 6 contains something different. In fact it is only the most-significant bit that is of consequence and as this is set to a '0', a positive mantissa is indicated. Just to verify this, change 20030 in program 8.8, to read

```
20030 B =-1400
```

On re-running, byte 6 of the FAC will now read 128+47, i.e. its MSB has been reset to a '1'.

Those earlier subroutines that required faith can now be tested using Programs 8.8 and 8.9. Take Program 8.4. This, it was said, converted an integer into floating-point and loaded it into FAC. Program 8.9 below tests this by printing out the contents of FAC; it uses 1400 ($0578) as the integer, i.e. LSB=$78=120 and MSB=$05=5.

They are loaded into Y (LDY #120) and A (LDA #5)

PROGRAM 8.9

```
LDY #120
LDA #5
JSR 58098
LDX #6
LDA 157,X
STA 1152,X
DEX
BPL 777
RTS
```

CONVERT INTEGER TO FLOATING POINT

Program 8.8 should then be modified so as not to put 1400 into FAC, i.e. line 20030 should read

```
20030 B=1        (or any number other than 1400)
```

The suite of programs can then be executed using a RUN 20000 and will display the value of A as 1400.

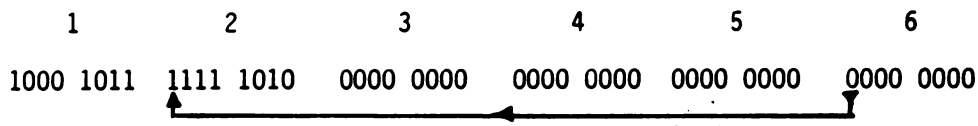
Doubters will also, most probably, wish to see Program 8.5 run, i.e. to load floating-point data from memory. This is done in Programs 8.10 and 8.11 where the number +2000 is used. However, a slight complication arises here, as floating-point numbers are stored in a slightly different format in memory from those in FAC and ARG. Taking +2000, for instance, this is stored in FAC and ARG as 139 250 0 0 0 0, or:

FAC ARG location	157 165	158 166	159 167	160 168
	139 1000 1011	250 1111 1010	0 0000 0000	0 0000 0000

FAC ARG location	161 169	162 170
	0 0000 0000	0 0000 0000

However, when storing large quantities of data, a more economical format is used which requires five bytes only. This is known as MELPT (Memory FloaTing PoinT) format; the six byte format used in FAC and ARG is known as FLPT. Byte 6 of FLPT is very wasteful as only bit 7 of byte 6 (memory location 102 or 110) is really needed. Obviously, to be able to dispense with byte 6 we have to find a way of storing bit 7 as it gives the sign of the mantissa, so one spare bit must be found elsewhere. Such a spare location exists in byte 2 on its left-most bit, as during normalisation of the mantissa the decimal point is moved until this bit is filled by a '1'. As it should always be set to '1' then the BASIC interpreter doesn't really need to read it to discover that it is '1' and therefore its location can be used for another purpose. The BASIC interpreter can be set up to assume that this is a '1' and automatically make an allowance made for this by adding a 1 to the left of the mantissa when using this. This redundant bit, therefore, can be and is used to store the sign bit, i.e. a '0' for positive and a '1' for negative, and byte 6 can be dispensed with, giving us the MFLPT format. The representation of +2000 in the alternative floating-point formats then becomes:

a) in FLPT format (as used in FAC and ARG):



b) in MFLPT format (as used for storing variables):

1	2	3	4	5
1000 1011	0111 1010	0000 0000	0000 0000	0000 0000
139	122	0	0	0

To use the "load FAC from memory" subroutine the variable data has to be loaded into memory, and this is most easily done using a direct POKE program - program 8.10 below:

PROGRAM 8.10

```
POKE 853,139
POKE 854,122
POKE 855,0
POKE 856,0
POKE 857,0
```

The machine code program to use this subroutine sets a pointer to the data in 853_{10} (0385_{16}) onwards and then transfers it with a JSR $EAF9.

Program 8.11 should be executed by means of the BASIC program 8.8.

PROGRAM 8.11

```
LDA #85
LDY #3
JSR $EAF9
LDX #6
LDA 157,X
STA 1152,X
DEX
BPL 777
RTS
```

TRANSFER DATA FROM MEMORY TO FAC

When run, program 8.11 will load the floating-point representation of +2000 into FAC and print this onto the screen. BASIC program 8.8 will then decode the graphic data and print this out in numbers.

Naturally ARG has a similar facility for transferring data, this being located at $E9E3 (59875).

Once the data is loaded into FAC or ARG a facility exists for copying from one to the other, that for copying ARG into FAC residing at $EB63 (60243).

A similar subroutine copies the other way, i.e. from ARG to FAC, this being located at $BC0C (60259).

> EXERCISE 8.1
>
> Write a suite of programs to load 2000 into ARG from memory and then print out the contents of this from 1552 onwards. Then transfer ARG into FAC and again print out the contents, this time from 1680 onwards.
>
> One possible answer is given in Chapter 9.

Floating-Point Subroutines

Having got numbers into FAC and ARG, use can be made of the built-in subroutines that offer the opportunity to handle these six-byte monsters with reasonable ease. A comprehensive list of these addresses is provided in Appendix 3.

A further caveat must be given along with the advice to try these. One must know where these subroutines pick up their data from and where they deposit the result, if they are to be used safely. Many of them start with a short initialisation section that prepares the data and deposits it in the right place for action. The subroutine that moves data from memory into ARG, for instance, starts by transferring its data address into $IF (31) and $23 (35) from the accumulator and Y register, hence when started at $E9E3 it expects to find the data address in A and Y, or a few bytes later with its address in $22 and $23. As an interesting exercise it may be of use to disassemble this subroutine and try and follow through the various stages. This can be done by selecting 'L' for list at the MENU and then listing from 59875.

These subroutines are just collections of 6502 instructions, much as you should be writing by now. However, they are very cleverly written and use bits of other subroutines to save space, and it is these JSR and JMP jaunts that you may find difficult to follow.

EXERCISE 8.2

Write a program suite to input the numbers 1.047 and 4038.22 into a machine code program. Multiply these together and then take the square-root of the product. Print out this answer on the screen from BASIC.

Second thoughts!!....

This is not as easy as it seems; the original plan looked like this:

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 INPUT B
20040 A = USR (B)
```

puts B into FAC

```
768 JSR 59306 (FAC - ARG)
    RTS
```

```
20050 POKE 11,12
20060 POKE 12,3
20070 INPUT D
20080 C = USR (D)
```

puts D into FAC

```
780 JSR $E982 (MULT)

783 JSR $EE8D

786 RTS
```

```
20090 PRINT "C= ";C
20100 END
```

That was the plan; however, it won't work! It fails because the program suite assumes that the FAC'S contents stay put while they are, in fact, constantly changing as BASIC runs a program. Following line 20040 FAC contains B and after the JSR (FAC-ARG) both FAC and ARG contain B. However, in executing lines 20050 to 20080 the operating system utilises FAC and ARG, and thus changes the contents of these. Thus when JSR(MULT) is called, the contents of FAC and ARG that are multiplied together are not those expected.

The problem can only be overcome by saving ARG in memory while returning to BASIC. This is done by means of the subroutine at $EB27. This subroutine copies the contents of ARG into the five bytes of memory starting at the address stored in $85 and $86. Program 8.15 illustrates this, putting ARG into $0384 onwards.

PROGRAM 8.15

```
LDA #132   } load LSB of address - 132
STA $85    } ($84) into $85.

LDA #3     } load MSB of address - 3
STA $86    }  ($03) into $86.

JSR $EB27
RTS
```

COPY ARG INTO MEMORY

The action of this can be checked by running the direct program: FOR X=0 TO 5 : PRINT PEEK(900+X);: NEXT X

To use this subroutine in Exercise 8.2, it is necessary to reload the data into ARG. This is done by the subroutine at $E9E3 (59875).

In order to operate, the subroutine needs to be told where to find the data and this is done by loading the address of the first byte of data into the accumulator (LSB) and the Y register (MSB). Thus a "reload ARG" program to recover data from 900 onwards would look like:

PROGRAM 8.16

```
LDA #132   Load LSB of address.
LDY #3     Load MSB of address.
JSR $E9E3  Load ARG from memory.
RTS
```

Perhaps now is a suitable time to try Exercise 8.2 One possible answer is given in Chapter 9.

Other Available Subroutines

Appendix 3 lists altogether 17 subroutines which have been found most useful when dealing with floating-point numbers from machine code. Those not yet discussed are covered below:

Addition

Using the subroutine at $E7C1 (47201), the floating-point numbers in FAC and ARG are added together and the sum loaded into FAC. This is demonstrated in programs 8.17 and 8.18.

PROGRAM 8.17

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 INPUT B
20040 A = USR (B)
20050 POKE 11,12
20060 POKE 12,3
20070 INPUT D
20080 C = USR (D)
20090 PRINT "C= ";C
20100 END
```

PROGRAM 8.18

```
LDA #132
STA 133
LDA #3
STA 134
JSR $EB27    Store FAC in memory
RTS

LDA #132     Retrieve data from
LDY #3       memory, store in ARG.
JSR $E9E3
JSR $E7C1    Addition subroutine
RTS
```

ADD TWO FLOATING-POINT NUMBERS

When the program at 20000 is run, it requests two inputs and then prints out the sum of these.

Subtraction

Programs 8.17 and 8.18 can be used to demonstrate this by inserting the subroutine $E7AA (59306) at 788 and 789 i.e.

PROGRAM 8.19 (part)

```
787 JSR $E7AA
```

This can be run by a RUN 20000 and will print out the answer. It subtracts the second input from the first.

Division

Once again we may use programs 8.17 and 8.18 to demonstrate the use of the Division routine at $EA69 (60009) - replace 788 and 789 as before.

PROGRAM 8.20 (part)

```
787 JSR $EA69
```

and RUN 20000. The program divides the first input by the second.

Exponentiation

The exponentiation routine is at $EE97 (61079) - so replace 788 and 789 once again.

PROGRAM 8.21 (Part)

```
787 JSR $EE97
```

and RUN 20000. The program raises the first inputted number to the power of the second input.

Other routines need only one input and operate on this alone; these are:

Log

The subroutine is at $E941 (47594) and computes the natural logarithm or $\log_e$ (log to the base e). Program 8.22 demonstrates this subroutine in use.

PROGRAM 8.22

```
768  JSR $E941
771 RTS
```

It is called by program 8.23

PROGRAM 8.23

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 INPUT B
20040 A = USR (B)
20050 PRINT "A= ";A
20060 END
```

A RUN 20000 activates both routines and prints out $\log_e$ of the input value.

The three functions SIN $EFF1 (61425)
COS $EFEA (61418)
and TAN $F03A (61498)
can be incorporated into a program suite such as 8.22/23. In each case, the input value is entered in radians and the computed function is returned in FAC.

Appendix 3 contains a very comprehensive list of the many subroutines which exist within the ROM, and I suspect that you find the list somewhat bewildering at first sight. As you develop more experience, however, I feel sure that you will find the list more and more useful, so don't be put off.

However, when using these built-in subroutines, the major problem lies in knowing exactly how to integrate these into any particular program as the subroutines' start-points are well documented. These problems broadly revolve around where the subroutine gets its data from and where it deposits its results. On the latter point, as will have been seen, the FAC is a common place in which to place results.

As to the linking in of data, the first few lines of a subroutine should give the clue. First, these lines should be disassembled using the assembler LIST function. Next these should be examined for the first use of A, X and Y as once these are re-loaded in the subroutine, any data originally in them will have been destroyed. If addresses are involved certain zero-page locations are popular such as $60/$62 (96/97) as well as $85/$86 (133/134). Appendix 3 also contains a list of the locations in the zero-page and following three pages together with a brief description of their uses.

Similarly, the deposition of processed data will be revealed by the last few instructions and if FAC and ARG are involved, their memory locations should feature. The end of the subroutine is usually an RTS instruction but it might be a JMP instruction to another of the ROM subroutines, the actual return being the result of the'called' routines RTS.

Finally, if logic (and all else!) fails there's always the 'shoot or bust' technique left, i.e. just trying it. If this is attempted, keep good records as constant reloading after crashes can be time-consuming. Of course, after a crash the machine-code program is not lost provided control can be regained by use of the reset key. However, if you have to resort to COLD BOOTING the Apple (<CTRL><OPEN APPLE><RESET>) to regain control then you will lose the machine code program in memory. The moral is to use the SAVE option of the Dr. Watson Assembler to save any lengthy routine before risking a run.

Remember that the subroutines in ROM are only machine-code like those you have produced and once disassembled should be quite comprehensible, if at times somewhat involved. Don't be afraid, therefore, to use things not detailed here! The 'experts' gained their exalted status by being inquisitive!

CHAPTER 9

Solutions to Exercises

Many of the exercises have more than one possible answer, so if you have a program that does the same thing as the answer in this chapter it is equally correct.

CHAPTER 1

EXERCISE 1.1 (PAGE 1-11)

```
START ADDRESS? 768

LDA # 1
STA 1024
RTS
END
```

EXERCISE 1.2 (PAGE 1-11)

```
START ADDRESS? 768

LDA # 198
STA 1024
LDA # 210
STA 1025
LDA # 197
STA 1026
LDA # 196
STA 1027
RTS
END
```

EXERCISE 1.3 (PAGE 1-12)

```
START ADDRESS? 768

LDA #216
STA 1024
STA 1063
STA 2000
STA 2039
RTS
END
```

EXERCISE 1.4 (PAGE 1-15)

```
START ADDRESS? 768

LDA #218
LDX #193
STA 900
TXA
LDX 900
STX 1024
STA 2039
RTS
```

EXERCISE 1.5 (PAGE 1-16)

```
START ADDRESS? 768

LDA #161
LDX #170
LDY #197
STX 900
TAX
TYA
LDY 900
STA 1024
STA 1063
STX 2000
STY 2039
RTS
```

CHAPTER 2

EXERCISE 2.1 (PAGE 2-6)

(i)

```
START ADRESS? 768

LDA #3
JSR 900
STA 1024
RTS
END
```

(ii)

```
START ADDRESS? 900

STA 950
ADC 950
RTS
END
```

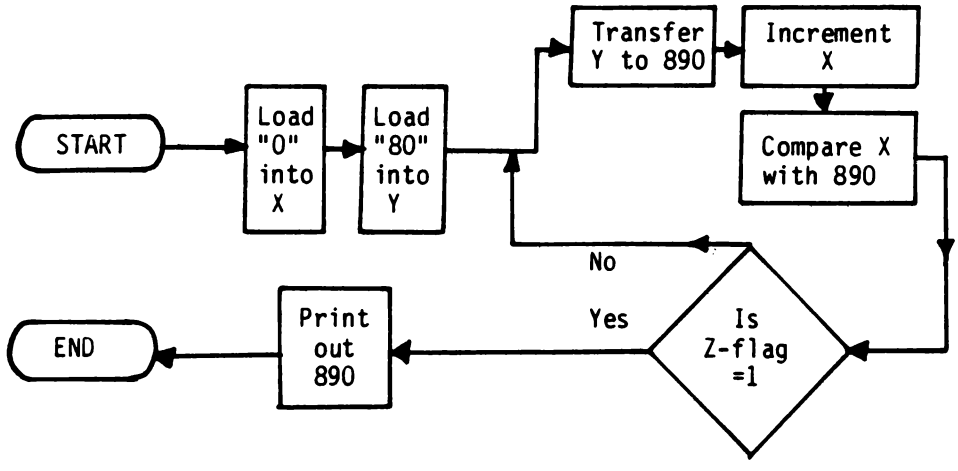
EXERCISE 2.2 (PAGE 2-8)

```
START ADDRESS? 768

LDY #100
DEY
BEQ 776
JMP 770
STY 1024
RTS
END
```

EXERCISE 2.3 (PAGE 2-11)

```
START ADDRESS? 768

LDA #191
STA 890
LDY #0
INY
CPY 890
BEQ 784
JMP 775
STY 1034
RTS
END
```

EXERCISE 2.4 (PAGE 2-13)



```
INX                 Increment X.
CPX 900             Compare X with 900.
BNE 777             Branch if not equal.
RTS                 Return from subroutine.
END
```

```
START ADDRESS? 768

LDX #0
LDY #80
STY 890
INX
CPX 890
BNE 775
STX 1024
RTS
END
```

CHAPTER 3

EXERCISE 3.1 (PAGE 3-2)

The LDX # instruction at 768 must be changed to a LDY #. From table 1 (Appendix 1) the object code for this is 160, so a

```
POKE 768,160
```

does the trick.

Tne STA...,X at 772 has to be changed to a STA...,Y (object code 153)

i.e. `POKE 832,153`

Lastly, the DEX at 775 needs to be replaced by a DEY (object code 136)

i.e. `POKE 775,136`

Thus the final program reads:-

```
LDY #100
LDA #218
STA 1023,Y
DEY
BNE 772
```

EXERCISE 3.2 (PAGE 3-2)

```
LDX #100          Load X with '100'.
STX 900           Store X in 900.
LDX #0            Load X with '0'.
LDA #170          Load A with 170 (asterisk).
STA 1024,X        Store A in (1024 + X)
```

```
INX             Increment X,
CPX   900       Compare X with 900
BNE   777       Branch if not equal
RTS             Return from subroutine.
END
```

CHAPTER 4

EXERCISE 4.1 (PAGE 4-7)

1807_{16} = 1 x 4096 + 8 x 256 + 0 x 16 + 7 x 1
= 4096 + 2048 + 07
= 6151_{10}

$2AFA_{16}$ = 2 x 4096 + 10 x 256 + 15 x 16 + 10 x 1
= 11002_{10}

i.e. $6151_{10} + 11002_{10} = 17153_{10}$

Program to add

```
START ADDRESS? 768

CLC
CLD
LDA #$07
ADC #$FA
STA 1026
LDA #$18
ADC #$2A
STA 1024
RTS
```

Screen display gives answer of - and A, i.e. 67_{10} and 01_{10}.

Converting this to hex gives:

67_{10} = 4 x 16 + 3 x 1 = $43

01_{10} = 0 x 16 + 1 x 1 = $01

Thus 67_{10} and 01_{10} = $4301
= 4x4096+3+256+0x16+1

= 17153_{10}

Cross-checking this by hex addition:

```
  1807
+ 2AFA
------
= 4301
```

EXERCISE 4.2 (PAGE 4-8)

```
START ADDRESS? 768
LDA #32
STA 890
LDA #88
STA 891
LDA #3
STA 892
LDA #2
STA 893
CLD
SEC
LDA 890
SBC 891
STA 1034
LDA 892
SBC 893
STA 1035
RTS
```

Screen display gives answer of H inverse @, i.e. 200 and 0.

EXERCISE 4.3 (PAGE 4-8)

```
CLD          Clear decimal to make sure in
             binary mode.
CLC          Clear carry flag.

LDA #$2C     Load LSB 1.                    } Add LSBs
ADG #$90     Add LSB 2 to A.                } together
STA 900      Sum of LSBS.

LDA #$01     Load MSB 1.                    } Add MSBs
ADC #$01     Add MSB 2                      } together
STA 901      Sum of MSBs                    } incl. carry
                                            } (if there)

SEC          Set carry in preparation       }
             for subtraction.               }
LDA 900      Load LSB 1+2 (of 700)          }   Subtract
SBC #$F4     Subtract LSB 3                 } LSBs
             from LSB 1+2.                  } and
STA 1041     Display sum of LSBs.           } display.

LDA 901      Load MSB 1+2 (of 700)          }   Subtract
SBC #$01     Subtract MSB 3                 } MSBs
             from MSB 1+2.                  } and
STA 1040     Display sum's MSB.             } display.

RTS
```

EXERCISE 4.4 (PAGE 4-14)

A = 1 B = 0 C (output) = 0
A = 0 B = 1 C = 0
A = 1 B = 1 C = 1

EXERCISE 4.5 (PAGE 4-14)

$149_{10} = 10010101_2$
$52_{10} = 00110100_2$
149 AND 52 = 00010100_2

$= 16 + 4$
$= 20_{10}$

EXERCISE 4.6 (PAGE 4-19)

	128	64	32	16	8	4	2	1
100 =	0	1	1	0	0	1	0	0
87 =	0	1	0	1	0	1	1	1
75 =	0	1	0	0	1	0	1	1
99 =	0	1	1	0	0	0	1	1
57 =	0	0	1	1	1	0	0	1
94 =	0	1	0	1	1	1	1	0
27 =	0	0	0	1	1	0	1	1

```
LDA #100
AND #87
STA 1024
RTS
```

When run gives a flashing 'D' (68_{10})

ii)

75	0	1	0	0	1	0	1	1
OR 27	0	0	0	1	1	0	1	1
=	0	1	0	1	1	0	1	1

$= 01011011$
$= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 1 = 91_{10}$

```
LDA #75
EOR #27
STA 1024
RTS
```

When run gives a flashing left square bracket (91_{10})

iii)

99	0	1	1	0	0	0	1	1
EOR 57	0	0	1	1	1	0	0	1
=	0	1	0	1	1	0	1	0

= 01011010

= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 0 = 90_{10}

When run gives a flashing 'Z' (90_{10})

iv)

100	0	1	1	0	0	1	0	0
AND 87	0	1	0	1	0	1	1	1
=	0	1	0	0	0	1	0	0
EOR 94	0	1	0	1	1	1	1	0
=	0	0	0	1	1	0	1	0

= 00011010

= 0 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = 26_{10}

```
LDA #100
AND #87
EOR #94
STA 1024
RTS
```

When run gives an inverse 'Z'

EXERCISE 4.7 (PAGE 4-21)

```
LDA #134
AND #15
STA 1025
LDY #4
LDA #134
LSR
DEY
BNE 779
STA 1024
RTS
```

When run this gives an output of an inverse H and F (i.e. 86)

EXERCISE 4.8 (PAGE 4-25)

```
LDX #3
STX 901
LDX #4
STX 902
LDY #8
LDA #0
CLC
LSR 902
BCC 792
CLC
ADC 901
ASL 901
DEY
BNE 782
STA 1024
RTS
```

When run this should print an inverse L (12) in 1024

CHAPTER 5

EXERCISE 5.1 (PAGE 5-4)

	In assembly language	In disassembled assembly language
	LDX #140	LDX #140
	JMP LOOP3	JMP 797
*LOOP1	LDA #165	LDA #165
	STA 1183,X	STA 1183,X
	DEX	DEX
	BNE LOOP1	BNE 773
	JMP END	JMP 810
*LOOP2	LDA #166	LDA #166
	STA 1023,X	STA 1023,X
	DEX	DEX
	BNE LOOP2	BNE 784
	LDX #120	LDX #120
	JMP LOOP1	JMP 773
*LOOP3	LDA #170	LDA #170
	STA 1303,X	STA 1303,X
	DEX	DEX
	BNE LOOP3	BNE 797
	LDX #160	LDX #160
	JMP LOOP2	JMP 784
*END	RTS	RTS

CHAPTER 7

EXERCISE 7.1 (PAGE 7-5)

POKE 769,160 (If program starts at 768)

The program should read:-

```
LDA #160
STA 900
JSR 64780
JSR 65005
CMP 900
BNE 773
LDA #141
JSR 65005
RTS
```

EXERCISE 7.2 (PAGE 7-11)

```
LDY #4
TXA
PHA
PHP
JSR $FDOC
JSR $FDED
PLP
PLA
TAY
TAY
DEY
BNE 770
RTS
END
```

In the above program, SR could have been loaded into stack before the Y register (via the accumulator), as long as it was then retrieved first, ie. LIFO operation.

CHAPTER 8

EXERCISE 8.1 (PAGE 8-18)
The data +2000 is to be stored from 900_{10} (0384_{16}) onward. This data is: 139 122 0 0 0 . It can be entered using:

PROGRAM 8.12

```
POKE 900,139
POKE 901,122
POKE 902,0
POKE 903,0
POKE 904,0
```

PROGRAM 8.13

```
LDA #132     Set up address for subroutine .
LDY #3       03(10) =03(10)
JSR $E9E3    Put data into ARG from memory.
LDX #6
LDA 156,X
STA 1551,X   Print FAC on screen.
DEX
BNE 777
JSR #EB53    Transfer data ARG ⟶ FAC.
LDX #6
LDA 156,X
STA 1679,X
DEX
BNE 790
RTS
```

TRANSFER ARG TO FAC

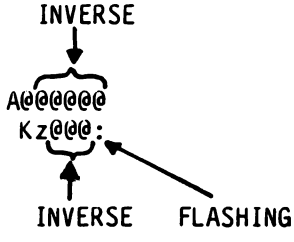
The BASIC program is:

PROGRAM 8.14

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 B=1
20040 A= USR (B)
20050 PRINT "A="A
20060 FOR X = 0 to 5
20070 PRINT PEEK (1424 + X);" ";
20080 NEXT X
20090 PRINT
20100 FOR X = 1 TO 61679
20110 PRINT PEEK (1624 + X);" ";
20120 NEXT X
20130 END
```

When run this should print out:

```
A= 2000
   129 128 0 0 0 0
   139 250 0 0 0 122
```



The display may differ between Apple IIs and IIes.

EXERCISE 8.2 (PAGE 8-19)

```
20000 HOME
20010 POKE 11,0
20020 POKE 12,3
20030 INPUT B
20040 A=USR(B)
20050 POKE 11,8
20060 POKE 12,3
20070 INPUT D
20080 C=USR(D)
20090 PRINT "C="C
20100 END
```

```
LDY #132
LDY #3
JSR $EB2B
RTS
LDA #132
LDY #3
JSR $E9E3
JSR $E982
JSR $EE8D
RTS
```

APPENDIX 2

EXERCISE A2.1 (PAGE A2-2)

i) $0000\ 0011_2 = 0+0+0+0+0+0+2+1 = 3_{10}$

ii) $0000\ 0100_2 = 0+0+0+0+0+4+0+0 = 4_{10}$

iii) $1000\ 0000_2 = 128+0+0+0+0+0+0+0 = 128_{10}$

iv) $1000\ 0011_2 = 128+0+0+0+0+0+2+1 = 131_{10}$

v) $1011\ 0111_2 = 128+0+32+16+0+4+2+1 = 183_{10}$

vi) $0111\ 0011_2 = 0+64+32+16+0+0+2+1 = 115_{10}$

EXERCISE A2.2 (PAGE A2-6)

i) $0009_{16} = 0x409+0x256+0x16+9x1 = 0+0+0+9 = 9_{10}$

ii) $0013_{16} = 0x4096+0x256+1x16+3x1 = 0+0+16+3 = 19_{10}$

iii) $00A5_{16} = 0+0+10x16+5x1 = 160+5 = 165_{10}$

iv) $0AAE_{16} = 0+10x256+10x16+14x1 = 2560+160+14 = 2734_{10}$

v) $000E_{16} = 0+0+0+14 = 14_{10}$

vi) $011A_{16} = 0+256+16+10 = 282_{10}$

vii) $00EA_{16} = 0+0+14x16+10 = 224+10 = 234_{10}$

viii) $FOA3_{16}$ = 15x4096+0+10x16+3
= 61440+160+3
= 61603_{10}

EXERCISE A2.3 (PAGE A2-8)

i) 4_{10} = 0100_2(BCD)

ii) 10_{10} = 1x10+0

iii) 77_{10} = 7x10+7
= 0111 0111_2(BCD)

iv) 97_{10} = 9x10+7
= 1001 0111_2(BCD)

v) 53_{10} = 5x10+3
= 0101 0011_2(BCD)

vi) 102_{10} = 1x100+0x10+2x1
= 0001 0000 0010_2(BCD)

vii) 953_{10} = 9x100+5x10+3x1
= 1001 0101 0011_2(BCD)

viii) 2579_{10} = 2x1000+5x100+7x10+9x1
= 0010 0101 0111 1001_2(BCD)

EXERCISE A2.4 (PAGE A2-8)

i) 0000 0001_2(BCD) = 0x10+1x1
= 1_{10}

ii) 0000 1001_2(BCD) = 0x10+9x1
= 9_{10}

iii) 0001 0101_2(BCD) = 1x10+5x1
= 15_{10}

iv) 0010 0000_2 = 2x10+0x1
= 20_{10}

v) 0100 1001_2(BCD) = 4x10+9x1
= 49_{10}

vi) $1010\ 0011_2$(BCD)

*** This is not a valid BCD number as the first nybble, $1010_2 = 10_{10}$, i.e. is greater than allowed in BCD.

vii) $1001\ 0111_2$(BCD) = $0x10+7x1$
$= 97_{10}$

viii) $1000\ 1000_2$(BCD) = $8x10+8x1$
$= 88_{10}$

APPENDIX 1

The 6502 Instruction Set

Abbreviations used:
.. means 8 bit number (1 byte operand)
... means 16 bit number (2 byte operand)
!!! means 16 bit address converted by the assembler to an 8 bit 2's complement displacement.

A = Accumulator
X = X register
Y = Y register

S = Set (to 1)
C = Clear (to 0)
? = Condition according to data.

ADC ADd specified contents to accumulator with Carry: store answer in accumulator; condition negative, overflow zero and carry flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ADC ...	109	6D	2	4	Absolute
ADC #	105	69	1	2	Immediate
ADC (..,X)	97	61	1	6	Indirect with X
ADC (..),Y	113	71	1	5*	Indirect with Y
ADC ...,X	125	7D	2	4*	Indexed with X
ADC ...,Y	121	79	2	4*	Indexed with Y
ADC ..	101	65	1	3	Zero-page
ADC ..,X	117	75	1	4	Zero-page indexed with X

* Plus 1 cycle if page boundary crossed.
To add without carry, clear carry flag (CLC) before ADC.
ADC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

AND — AND specified contents with accumulator: store answer in accumulator: condition negative and zero flags according to result.

MNEMONIC	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
AND ...	45	2D	2	4	Absolute
AND #	41	29	1	2	Immediate
AND (..,X)	33	21	1	6	Indirect with X
AND (..),Y	49	31	1	5*	Indirect with Y
AND ...X	61	3D	2	4*	Indexed with X
AND ...,Y	57	39	2	4*	Indexed with Y
AND ..	37	25	1	3	Zero-page
AND ..,X	53	35	1	4	Zero-page indexed with X

* Plus 1 cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	0
1	0	1

ASL — Arithmetic Shift Left of specified contents: bit 7 put into carry, '0' into bit zero; condition negative and zero flags according to result.

MNEMONIC	OP-CODE DEC	OP-CODE HEX	NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
ASL ...	14	03	1	6	Absolute
ASL	10	0A	0	2	Accumulator
ASL ...,X	30	1E	2	7	Indexed with X
ASL ..	6	06	2	5	Zero-page
ASL ..,X	22	16	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

BCC Branch on Carry Clear: Test carry flag, if clear (C=0) branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BCC !!!	144	90	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: No effect.

BCS Branch on Carry Set: test carry flag, if set (C=1), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BCS !!!	176	B0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BEQ Branch on result EQual to zero: test Z-flag, if set, (Z=1), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BEQ !!!	240	F0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BIT AND specified BITs with accumulator: A remains unaltered; set Z (=1) if bits match, transfer bits 6 and 7 of specified data to V and N flags respectively; condition zero flag according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BIT ...	44	2C	2	4	Absolute
BIT ..	36	24	1	3	Zero-page

N	V	B	D	I	Z	C
Bit 7	Bit 6	-	-	-	?	-

BMI Branch on result MInus: test N flag if set (N=1), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BMI !!!	48	30	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BNE Branch on result Not Equal to zero: test Z flag if not set (Z=0), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BNE !!!	208	D0	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BPL Branch on result PLus: test N flag, if not set (N=0), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BPL !!!	16	10	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BRK BReaK into interrupt: initiate interrupt sequence; Save PC+2 on stack; set B flag (B=1); save PSW on stack; load interrupt vectors (FFFE and FFFF) into PC. Set I flag (I=1).

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BRK	0	0	0	7	Implied

N	V	B	D	I	Z	C
-	-	S	-	S	-	-

BVC Branch on oVerflow Clear: test overflow flag, if not set (V=$\overline{0}$), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BVC !!!	80	50	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BVS Branch on oVerflow Set: test overflow flag, if set (V=1), branch relative.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BVS !!!	112	70	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

CLC CLear Carry flag: load '0' into carry flag (C=0).

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLC	24	18	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	C

CLD CLear Decimal flag: load '0' into decimal flag.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MOE
	DEC	HEX			
CLD	216	D8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	C	-	-	-

CLI CLear Interrupt disable flag: load '0' into interrupt flag (I=0).

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLI	88	58	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	C	-	-

CLV CLear oVerflow flag: load '0' into overflow flag (V=0).

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLV	184	B8	0	2	Implied

N	V	B	D	I	Z	C
-	0	-	-	-	-	-

CMP CoMPare specified data with accumulator: subtract data from accumulator, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CMP ...	205	CD	2	4	Absolute
CMP #(..,X	201	C9	1	2	Immediate
CMP (..,X)	193	C1	1	6	Indirect with X
CMP (..),Y	209	D1	1	5*	Indirect with Y
CMP ...,X	221	DD	2	4*	Indexed with X
CMP ...,Y	217	D9	2	4*	Indexed with Y
CMP ..	197	C5	1	3	Zero-page
CMP ..,X	213	D5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPX ComPare specified data with X register: subtract data from X register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CPX ...	236	EC	2	4	Absolute
CPX #	224	E0	1	2	Immediate
CPX ..	228	E4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPY ComPare specified data with Y-register: subtract data from Y-register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CPY ...	204	CC	2	4	Absolute
CPY #	192	C0	1	2	Immediate
CPY ..	196	C4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

DEC DECrement specified memory contents by one, store result in specified memory location; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEC ...	206	CE	2	6	Absolute
DEC ...,X	222	DE	2	7	Indexed with X
DEC ..	198	C6	1	5	Zero-page
DEC ..,X	214	D6	1	6	Zero-page, indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEX DEcrement contents of X-register: store result in X-register; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEX	202	CA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEY DEcrement contents of Y-register: store result in Y-register; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEY	136	88	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

EOR Perform Exclusive OR between accumulator and specified contents: store result in accumulator. Condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
EOR ...	77	4D	2	4	Absolute
EOR #	73	49	1	2	Immediate
EOR (..,X)	65	41	1	6	Indirect with X
EOR (..),Y	81	51	1	5*	Indirect with Y
EOR ...,X	93	5D	2	4*	Indexed with X
EOR ...,Y	89	59	2	4*	Indexed with Y
EOR ..	69	45	1	3	Zero-page
EOR ..,X	85	55	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	1
1	1	0
1	1	0

INC INCrement specified contents by one: store result in specified location; condition negative and zero page flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INC ...	238	EE	2	6	Absolute
INC ...,X	254	FE	2	7	Indexed with X
INC ..	230	E6	1	5	Zero-page
INC ..,X	246	F6	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INX INcrement X-register by one: store result in X-register; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INX	232	E8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INY INcrement Y-register by one: store result in Y-register; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INY	200	C8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

JMP JuMP to specified address (load specified address into program counter).

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
JMP ...	76	4C	2	3	Absolute
JMP (...)	108	6C	2	5	Indirect

FLAGS: no effect.

JSR Jump to SubRoutine at specified address: store program counter contents +2 on stack; load specified contents into program counter.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
JSR ...	32	20	2	6	Absolute

FLAGS: no effect.

LDA LoaD Accumulator with specified contents: condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDA ...	173	AD	2	4	Absolute
LDA #	169	A9	1	2	Immediate
LDA (..,X)	161	A1	1	6	Indirect with X
LDA (..),Y	177	B1	1	5*	Indirect with Y
LDA ...,X	189	BD	2	4*	Indexed with X
LDA ...,Y	185	B9	2	4*	Indexed with Y
LDA ..	165	A5	1	3	Zero-page
LDA ..,X	181	B5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDX LoaD X-register with specified contents: condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDX ...	174	AE	2	4	Absolute
LDX #	162	A2	1	2	Immediate
LDX ...,Y	190	BE	2	4*	Indexed with Y
LDX ..	166	A6	1	3	Zero-page
LDX ..,Y	182	B6	1	4	Zero-page indexed with Y

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDY Load Y-register with specified contents: condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDY ...	172	AC	2	4	Absolute
LDY #	160	A0	1	2	Immediate
LDY ...,X	188	BC	2	4*	Indexed with X
LDY ..	164	A4	1	3	Zero-page
LDY ..,X	180	B4	1	4	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LSR Perform Logical Shift Right of specified contents: load bit 0 into carry bit and a '0' into bit 7; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LSR ...	78	4E	2	3	Absolute
LSR	74	4A	0	2	Accumulator
LSR ...,X	94	5E	2	3	Indexed with X
LSR ..	70	46	1	6	Zero-page
LSR ..,X	86	56	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

NOP No OPeration; wait two cycles then continue.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
NOP	234	EA	0	2	Implied

FLAGS: no effect.

ORA Perform logical OR between Accumulator and specified contents: store result in accumulator; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ORA ...	13	0D	2	4	Absolute
ORA #	9	09	1	2	Immediate
ORA (..,X)	1	01	1	6	Indirect with X
ORA (..),Y	17	11	1	5*	Indirect with Y
ORA ...,X	29	1D	2	4*	Indexed with X
ORA ...,Y	25	19	2	4*	Indexed with Y
ORA ..	5	05	1	3	Zero-page
ORA ..,X	21	15	1	4	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	1
1	1	1
1	1	1

PHA Push Accumulator onto stack: update stack pointer; A remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHA	72	48	0	3	Implied

FLAGS: no effect.

PHP PusH Processor status word onto stack: update stack pointer; PSW remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHP	8	08	0	3	Implied

FLAGS: no effect.

PLA PuLl Accumulator from stack: update pointer; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PLA	104	68	0	4	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

PLP PuLl Processor status word from stack: update stack pointer; condition ALL flags according to PSW retrieved.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PLP	40	28	0	4	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

ROL RUtate Left one place, specified contents: load carry bit into bit 0 of specified data and bit 7 of these into carry flag; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ROL ...	46	2E	2	6	Absolute
ROL	42	4A	0	2	Accumulator
ROL ...,X	62	3E	2	7	Indexed with X
ROL ..	38	26	1	5	Zero-page
ROL ..,X	54	36	1	6	Zero-page indexed with X

ROL is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 7

ROR RUtate Right one place, specified contents: load carry bit into bit 7 of specified data and bit 0 of these into carry flag; condition negative and zero flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ROR ...	110	6E	2	6	Absolute
ROR	106	6A	0	2	Accumulator
ROR ...,X	126	7E	2	7	Indexed with X
ROR ..	102	66	1	5	Zero-page
ROR ..,X	118	76	1	6	Zero-page indexed with X

ROR is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 0

RTI ReTurn from Interrupt: retrieve PSW and PC from stack, update stack pointer.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
RTI	64	40	0	6	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

RTS ReTurn from Subroutine: retrieve PC from stack and increment by one, update stack pointer.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
RTS	96	60	0	6	Implied

FLAGS: no effect.

SBC SuBtract with Carry specified contents from accumulator: store answer in accumulator; condition negative, overflow, zero and carry flags according to result.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SBC ...	237	ED	2	4	Absolute
SBC #	233	E9	1	2	Immediate
SBC (..,X)	225	E1	1	6	Indirect with X
SBC (..),Y	241	F1	1	5*	Indirect with Y
SBC ...,X	253	FD	2	4*	Indexed with X
SBC ...,Y	249	F9	2	4*	Indexed with Y
SBC ..	229	E5	1	3	Zero-page
SBC ..,X	245	F5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.
To subtract without carry, set carry flag (SEC) before SBC. SBC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

SEC SEt Carry flag: C=1

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SEC	56	38	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	S

SED SEt Decimal flag: D=1

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SED	248	F8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	S	-	-	-

SEI SEt Interrupt disable flag: I=1

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SEI	120	78	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	S	-	-

STA STore Accumulator contents at specified address: A remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STA ...	141	8D	2	4	Absolute
STA (..,X)	129	81	1	6	Indirect with X
STA (..),Y	145	91	1	6	Indirect with Y
STA ...,X	157	9D	2	5	Indexed with X
STA ...,Y	153	99	2	5	Indexed with Y
STA ..	133	85	1	3	Zero-page
STA ..,X	149	95	1	4	Zero-page indexed with X

FLAGS: no effect.

STX STore contents of X-register at specified address: X remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STX ...	142	8E	2	4	Absolute
STX ..	134	86	1	3	Zero-page
STX ..,Y	150	96	1	4	Zero-page indexed with Y

FLAGS: no effect.

STY STore contents of Y-register at specified address: Y remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STY ...	140	8C	2	4	Absolute
STY ..	132	84	1	3	Zero-page
STY ..,X	148	94	1	4	Zero-page indexed with X

FLAGS: no effect.

TAX Transfer contents of Accumulator to X-register: A remains unaltered; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TAX	170	AA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TAY Transfer contents of Accumulator to Y-register: A remains unaltered; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TAY	168	A8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TSX Transfer contents of Stack-pointer to X-register: SP remains unaltered; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TSX	186	BA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXA Transfer contents of X-register into Accumulator: X remains unalted; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TXA	138	8A	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXS Transfer contents of X-register into Stack: X remains unaltered.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TXS	154	9A	0	2	Implied

FLAGS: no effect.

TYA Transfer contents of Y-register into Accumulator: Y remains unaltered; condition negative and zero flags according to data.

MNEMONIC	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TYA	152	98	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

6502 STATUS FLAG GUIDE

This guide lists all the 6502 flags along with the families of instructions that set them and the branch instructions that test their condition. As all members of the family condition the flags in the same way the whole family is referenced by the first three letters of the mnemonic, i.e. ADC refers to the Add with Carry family for all the possible addressing modes used with ADC.

NEGATIVE FLAG

Instruction to Condition				Instruction to Test
ADC	DEY	LSB	TAY	BMI
AND	EOR	ORA	TAY	BPL
ASL	INC	PLA	TSX	
CMP	INX	PLP	TXA	
CPX	INY	ROL	TYA	
CPY	LDA	ROR		
DEC	LDX	RTI		
DEX	LDY	SBC		

With BIT instruction, bit 7 of data is transferred into N-flag.

V - OVERFLOW FLAG

Instruction to Condition			Instruction to Test
ADC	CLV	RTI	BVC
BIT	PLP	SBC	BVS

BIT loads V with bit 6 of the specified memory location.

B - BREAK FLAG

Instruction to Condition			Instruction to Test
BRK	PLP	RTI	None

D - DECIMAL FLAG

Instruction to Condition				Instruction to Test
CLD	PLP	RTI	SED	None

I - INTERRUPT FLAG

Instruction to Condition				Instruction to Test
CLI	PLP	RTI	SEI	None

Z - ZERO FLAG

Instruction to Condition				Instruction to Test
ADC	DEC	LDA	ROL	BEQ
AND	DEX	LDX	ROR	BNE
ALS	DEY	LDY	RTI	
BIT	EOR	LSR	SBC	
CMP	INC	ORA	TAX	
CPY	INX	PLA	TAY	
CPX	INY	PLP	TXA	
			TYA	

C - CARRY FLAG

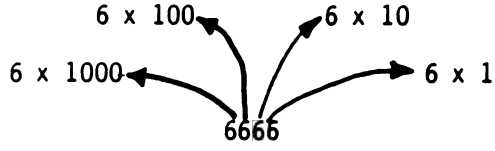
Instruction to Condition				Instruction to Test
ADC	CPX	ROL	SEC	BCC
ASL	CPY	ROR		
CLC	LSR	RTI		
CMP	PLP	SBC		

APPENDIX 2

Binary, Binary-Coded Decimal and Hexadecimal Notations

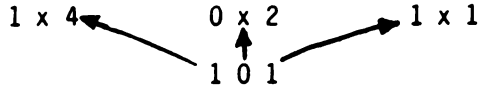
Counting systems in general use throughout the world use the decimal system and this has been developed to count up to and beyond 10 and also below the value 1. In this standard the digits to the left of a number are of greater value than those to the right. For instance, in the number 66, the first 6 has a value 10 times the second, i.e.

This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit in a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater by being multiplied by the base.

The computer, being basically electronic in operation, works better if it is told to only recognise two states, on or off or '0' and '1', and thus uses the Binary system - base 2. Thus, any number in binary consists simply of 0's and 1's, or electronically, zero volts (off) and some volts (on). To count past one, the binary system must resort to place-value notation and, as with other cases, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e. 4+0+1=5. Clearly the plethora of bases presents a problem when representing numbers as in base 10, '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers in that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

101_{10} = One hundred and one in base ten.

101_2 = Five in base two.

The present-day generation of personal computers (1984-style) use eight bit registers or memories and can thus represent numbers up to 11111111_2, i.e. in base 10:

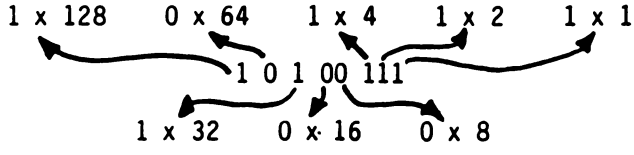
128	+64	+32	+16	+8	+4	+2	+1
1	1	1	1	1	1	1	1
128	64	32	16	8	4	2	1

$=255_{10}$

Digit Equivalent in base 10

Fig. A2.1

By way of example, let's take one more conversion - say, 10101011_2.



Thus 10100111 = 1x128 +0x64 +1x32 +0x16 +0x8 +1x4 +1x2 +1x1
= 128+32+4+2+1
= 167_{10}

Just to check your understanding, have a go at the following:

EXERCISE A2.1

Calculate the value of the following in base 10:-

i) 00000011_2
ii) 00000100_2
iii) 10000000_2
iv) 10000011_2
v) 10110111_2
vi) 01110011_2

Answers in Chapter 9.

If you remain unclear on this, or simply want to see it demonstrated, load and run the Binary/Hex tutor program which is included on the assembler disk. At the menu select 'H' for "Decimal, Binary and Hexadecimal". Then, when asked "What number do you wish to start at?", press "1" <return>. The screen will then display three rows of boxes, of which the top two are currently of interest. These represent a decimal number (marked "DEC") and a binary number (marked "BIN"). At this stage, they should contain the numbers 1_{10} and 1_2. The decimal number has three digits and thus has a capacity of 999_{10}, and the binary, with its eight binary digits (bits) will hold up to 255_{10}.

From this point, the program will simply count, every time that you press A, both the decimal and the binary boxes will index one. Try pressing the space bar once and the boxes should contain a 2_{10} and a 10_2. If you carry on indexing then you will see how binary counts. When you get to the stage where the decimal shows 15_{10} the binary should read 1111_2. Now index one further and the binary will change to 10000_2. One way of looking at this is to lay out the addition:-

```
 1111    A
+   1    B
```

On adding the 1 (A) to the 1 (B) this gives '2' i.e. 0, carry 1. This carry then produces another '0' plus another carry, and so on.

If you continued to press A long enough, then eventually the binary register would become full. However, this would take an awful lot of pressing, so we will take a short cut to this state of affairs. Instead of pressing A, press X instead. This will return you to the menu where you can select the 'H' option again. This time, when asked "What number do you wish to start at?" type a fairly high value which is less than 255, say 240. Off you go again until the binary register is full i.e. 1111111111_2. The addition of a further one, now, will clock all the binary register back to zeros and 256 will be lost. However, with the 6502, <u>all</u> is not lost as the 6502 has a carry flag that stores the fact that a carry has occurred. Clocking past 255_{10} with the Binary/Hex tutor will show this happening. This is a handy feature of the 6502 but it must not be relied on as more than a temporary store of the carry. The carry flag is just as easily reset as it is set to 1!

In order to make sure that you really understand the binary notation, you may wish to try some of the exercises which are provided by the BIN/HEX exercises. Select 'E' at the main menu. This will provide you with a menu of exercises and you can select '2' to try the exercises converting decimal numbers into binary or '5' to try converting them back again. Use the cursor control keys at the bottom right of the keyboard to move the cursor from one digit to the next. When you are satisfied that you have done enough, pressing X will take you back to the main menu.

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1_2 and 1_{10} there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255_{10}) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind-bending! Time to think ... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111_2 or 15_{10} and this requires, along with the 0, sixteen different symbols. The ones adopted for this job are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Decimal number
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Symbol

Fig. A2.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NYBBLES. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) - see Fig. A2.3.

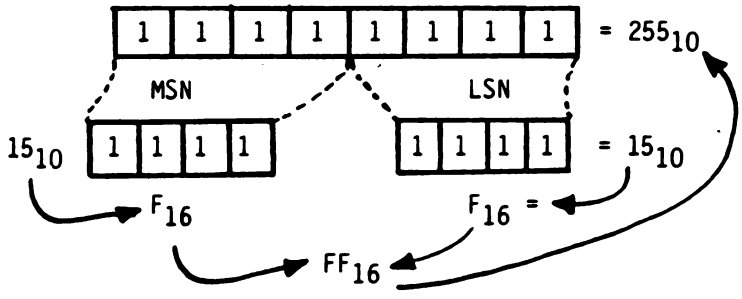


Fig. A2.3

The system described, which uses sixteen symbols is, of course, given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

You are now in a position to look at the Binary/Hex tutor program again. The third row of boxes, which we ignored last time round, contains the Hex numbers. While the counting is going on in the binary boxes, so it is in the Hex boxes also. The comparability between binary and HEX shows wherever a major carry occurs - take for instance 1111_2, 15_{10} or F_{16}: one index past this clocks the binary ones to zeroes and adds a one to the left, i.e. to 10000_2 or 10_{16}. These major points of correspondence occur at

$$1_2 = 1_{16} = 1_{10}$$

$$0001\ 0000_2 = 10_{16} = 16_{10}$$

$$0000\ 0001\ 0000\ 0000_2 = 100_{16} = 256_{10}$$

$$0001\ 0000\ 0000\ 0000_2 = 1000_{16} = 4096_{10}$$

Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together, e.g. FF_{16}=255, once again calls for place-value notation. This time, as the base is 16 the ratio between any place and its neighbour is 16.

The values, in base 10 of the places in hexadecimal are:

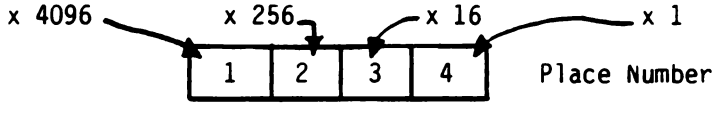


Fig. A2.4

Using Fig. A2.4 the way that $E92F_{16}$ makes up 59695_{10} is explained below in figure A2.5.

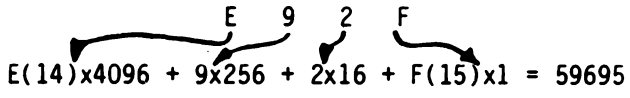


Fig. A2.5

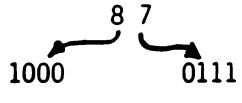
Now that hex is totally mastered(!) try the following; the first two are explained fully in Chapter 9.

> EXERCISE A2.2
>
> Calculate the value in decimal of the following:-
>
> i) 0009_{16} v) $000E_{16}$
> ii) 0013_{16} vi) $011A_{16}$
> iii) $00A5_{16}$ vii) $00EA_{16}$
> iv) $0AAE_{16}$ viii) $F0A3_{16}$
>
> Answers in Chapter 9.

Binary-Coded Decimal

As well as decimal, binary and hexadecimal notations, one other system is used in computing - binary-coded decimal. As its name suggests it is a hybrid form with elements from binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:



i.e. BCD = 1000 0111 (or in eight bits 10000111)

Fig. A2.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10}=1001_2$, thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A2.6 shows this, where 87_{10} is represented in BCD as 10000111_2. This can give rise to ambiguity in that 10000111_2 in binary is 135_{10}. To overcome this, BCD representations will be given the notation 10000111_2(BCD).

Using four bits of binary, it is possible to count up to 15_{10} (i.e. $1111_2=15_{10}$) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_2 and carries the 1 to the next nybble, i.e.

8_{10}	=	0000 1000	(base 2 BCD)
9_{10}	=	0000 1001	" " "
10_{10}	=	0001 0000	" " "
11_{10}	=	0001 0001	" " "

Fig. A2.7

It would probably be helpful at this point if you load and run the Binary/Hex tutor program again. This time, select 'B' at the main menu, and when asked "At what number..." enter a 1 <return>.

The display will then show three rows of boxes again but this time they will contain decimal, binary and BCD. If you press A as before, to index from '1', you will notice that up to 9_{10}, binary and BCD are identical. However, as you index from 9_{10} to 10_{10} keep an eye on the BCD box and you will see the 1 carried over to the most significant nybble. From 10_{10} upwards BCD becomes a true hybrid representing the decimal number in a binary form.

As the number increases, the uneconomical nature of BCD will become apparent as 99_{10} changes to 100_{10}. (As before, typing X will get you back to the main menu, which will allow you to restart at a value nearer to 99_{10}.) When 99_{10} indexes to 100_{10} you will see the BCD generate a carry from its most significant nybble to the carry flag.

As mentioned above, this carry is only a short term expedient and must be picked up at the earliest possible moment if it is not to be lost. The carry is generated on the BCD boxes at 99_{10} while the binary boxes will store up to 255_{10}. BCD is therefore fairly uneconomical in memory usage, but it has its uses in particular situations. In the past, microcomputers have always been dogged by their lack of memory and consequently BCD has been little used. However, the new generation of microcomputers have much larger memories and it is quite likely that BCD will be used much more frequently than it was in the past.

As you know all about BCD now! try the following:-

EXERCISE A2.3
Convert the following decimal numbers into BCD:

i)	4	v)	53
ii)	10	vi)	102
iii)	77	vii)	953
iv)	97	viii)	2579

Answers in Chapter 9.

EXERCISE A2.4

Convert the following BCD numbers into decimal:-

i)	0000 0001
ii)	0000 1001
iii)	0001 0101
iv)	0010 0000
v)	0100 1001
vi)	1010 0011
vii)	1001 0111
viii)	1000 1000

Answers in Chapter 9.

In the explanations given of the value of places in place-value notation a simplification was adopted in order to make these explanations clearer for our less mathematically inclined brethren. However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX 2.

With binary numbers it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact the multiplying factor is the base raised to the power of its place starting with zero at the left. i.e. in binary:

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Place
Previously stated multiplication factor

Mathematically more precise factor.

Thus the least significant bit is multiplied by 2^0 or 1. (If you are not sure of this try the direct program PRINT 2 0.) The next bit is multiplied by 2^1, and so on.

This rule holds for ANY base; let's apply it for hex, i.e. base 16:

Least significant bit factor = 16^0 = 1

2nd most significant bit factor = 16^1 = 16

3rd most significant bit factor = 1622 = 256

Most significant bit factor = 16^3 = 4096

APPENDIX 3

Memory Maps and Vectors

LABEL	ADDRESS HEX	DEC	DESCRIPTION
USRJMP	0A-0C	10-12	ISR jump vector
CHARAC	0D	13	Search character
ENDCHAR	0E	14	Flag: scan for quotes at end of string
COUNT	0F	15	Input buffer pointer/Number of subscripts
DIMFLG	10	16	Flag: Default array dimension
VALTYP	11	17	Data type flag: $00=numeric,#ff =string
INTFLG	12	18	Data type flag:$00=floating point, $80=integer
SUBFLG	14	20	Flag: Subscripts reference/User function call
INPFLG	15	21	Input flag: $00=INPUT,$40=GET, $98 = READ
COMPRTYP	16	22	Comparative result
WNDLFT	20	32	Left edge of text window
WNDWDTH	21	33	Width of text window
WNDTOP	22	34	Top of text window
WNDBTM	23	35	Bottom of text window
CH	24	36	Horizontal position of cursor
CV	25	37	Vertical position of cursor
GBASL	26	38	Base address for low-res line
GBASH	27	37	
BASL	28	38	Base address for text line
BASH	29	39	
BAS2L	2A	40	Used in scrolling
BAS2H	2B	41	
INVFLG	32	50	Controls of characters are printed normal, inverse or flashing
PROMPT	33	51	Prompt character for beginning of the line
YSAV	34	52	
YSAVI	35	53	Contents of the y register saved here by COUT
CSWL	36	54	Vector for output
CSWH	37	55	
KSWL	38	56	Vector for input
KSWH	39	57	
RNDL	4E	78	Random number - see KEYIN
RNDH	4F	79	

LABEL	ADDRESS HEX	DEC	DESCRIPTION
LINNUM	50,51	80,81	Line number found by FNDLIN/ general purpose
INDEX	5E,5F	94,95	Utility pointer area
INDEX2	60,61	96,97	Second utility pointer
RESHO	62-66	98-102	Floating point product of multiply and divide
TXTTAB	67,68	103,104	Pointer: start of BASIC text area
VARTAB	69,6A	105,106	Pointer: start of BASIC variables
ARYTAB	6B,6C	107,108	Pointer: start of BASIC arrays
SPDBYT	F1	241	Speed at which printing occurs
TRON	F2	242	Trace flag: if value greater than $7F then TRACE active
ORMSK	F3	243	
ONERRTXT	F4-F5	244-245	TXTPTR save for RESUME by HINDLERR
ONERRLIN	F6-F7	246-247	Line number for use by RESUME
REMSTK	F8	248	Value of stack saved before each statement
ANGLE	F9	249	Angle through which shape is rotated
	FA-FF	250-255	Unused by BASIC or DOS

Apple Memory Map

LABEL	ADDRESS	DESCRIPTION
STACK	0100-01FF	Area used by the 6502 stack
FBUFFR	0100-0110	Floating point output buffer
IN	0200	Start of input line buffer
BUF	0200-02FF	Input line number
WSDOS	03D0	Jump to warmstart DOS
CSDOS	03D3	Jump to coldstart DOS
TOFM	03D6	Jump to file manager
TORWTS	03D9	Jump to RWTS
FNDFMPAR	03DC	Locate file manager parameter list
FNDRWPAR	03E3	Locate RWTS parameter list
REPVEC	03EA	Replace KSW and CSW with DOS vectors
BRKV	03F0,03F1	Break vector
SOFTEV	03F2,03F3	Reset vector
PWREDUP	03F4	Reset vector check byte
AMPERV	03F5-03F7	Ampersand vector
LINE1	0400	Start of first line of the screen
	9600	Highest address available to BASIC with DOS and three buffers active
	9D00	Start of DOS
	9D00,9D01	Address of first DOS buffer
	9D02,9D03	Address of DOS keyboard intercept routine
	9D04,9D05	Address of DOS video intercept routines
	9D5A	Address of BASIC error handling routine
	9D5C	Address of BASIC warmstart
	AA53,AA54	Vector: CSW routine
	AA55,AA56	Vector:KSW routine
	AA60,AA61	Length of BLOADed program
	AA72,AA73	Start address of BLOADed progam
	BFFF	Highest address available to BASIC (48k)
KBD	C000	Keyboard. Data valid if greater than $7F
KBDSTRB	C010	Keyboard strobe
SPKR	C030	Speaker
TXTCLR	C050	Select graphics
TXTSET	C051	Select text
MIXCLR	C052	Select all graphics
MIXSET	C053	Select mixed text and graphics
LOWSCR	C054	Select primary page
HISCR	C055	Select secondary page

LABEL	ADDRESS	DESCRIPTION
LORES	C056	Select lo-res graphics
HIRES	C057	Select high-res graphics
PADDLO	C064	
PTRIG	C070	
	D000-F7FF	BASIC ROM
STMDSP	D000-D07F	BASIC command vector table
FUNDSP	D080-D0B1	BASIC function vector table
OPTAB	D0B2-D0CF	BASIC operators vector & priority table - 2 byte address & one priority
RESLST	D0D0-D25F	Table of BASIC keywords
ERRTAB	D260-D355	Table of error messages
BLTU	D393	Move block of memory up - check sufficient room then...
BLTUC	D39A	Move block (LOWTR) to (HIGHTR) -1 up to new block ending at (HIGHDS)-1
REASON	D3E3	Check address (A/Y) is lower than bottom of string space - if not....
OMERR	D410	Print "OUT OF MEMORY" IF NOT
ERROR	D412	Print error message indicated by (X)
INLIN	D52C	Input line into BASIC buffer
GDBUFS	D539	Mask out msb. replace $8D with $00
INCHR	D553	Get one character in A, mask off msb
RUN	D566	Run the program in memory
FNDLIN	D61A	Search BASIC text from start for line number in (LINNUM)...or...
FNDLNC	D61E	Search BASIC text from (A/X) for line number in (LINNUM) - if found: set C and (LINPTR) points to start of line - else clear C
SCRTCH	D64B	New command
CLEAR	D66A	
CLEARC	D66C	Clear variables and stack
STKINI	D683	Initialises stack pointer
STXTPR	D697	Initialise stack
LIST	D6A5	
FOR	D766	
NEWSTT	D7D2	Execute the next statement
RESTOR	D849	Set DATPTR to beginning of the program
ISCNYC	D858	Checks for a CONTROL-C, PRINTS BREAK IN <line nuber> if found

LABEL	ADDRESS	DESCRIPTION
CONT	D898	CONT command
PROGIO	D901	Set up A1 and A2 to save program
GOSUB	D921	
GOTO	D93E	GOTO command
DATA	D995	Move TXTPTR to the end of the statement
DATAN	D9A3	Calculate offset from TXTPTR to next ':' or eol (0)
REMN	D9A6	Calculate offset from TXTPTR to next eol (0)
REM	D9DC	
ADDON	D998	Add Y to TXTPTR
LINGET	DA0C	Read a line number (0 to 63999) into LINNUM
LET	DA46	Evaluate formula
CRDO	DAFB	Print a return
COPY	DAB7	Free the string temporarily
STROUT	DB3A	Print the string pointed to by Y,A
STRPTR	DB3D	Print the string pointed to by FACMO,FACLO
OUTSPC	D657	Print a space
QUTQST	DB5A	Print a question mark, "?"
OUTDO	DB5C	Print character in A
INPUT	DBB2	INPUT command
READ	DBE2	READ command
FRMNUM	DD67	Evaluate the formula at TXTPTR and put the result in FAC
CHKNUM	DD6A	Make sure FAC is a number
CHKSTR	DD6C	make sure FAC is a string
CHLVAL	DD6D	Check the result of the most recent FAC operation is a numeric or string variable. C is set for strings, clear for numeric

LABEL	ADDRESS	DESCRIPTION
FRMEVL	DD7B	Input and evaluate any expression in BASIC text. Sets VALTYP (00 if numeric, $FF if string) and ZINTFLG (00 if floating point, 80 if integer). If expression is numeric floating point, result is returned in FAC. If expression is numeric integer, result is returned in (FAC+3) in HI/LO format. If expression is string, then a pointer to the string descriptor is returned in (FAC+3), this is usually a copy of VARPNT. IN ADDITION, if expression is a simple variable, then VARNAM will be set to point to the first byte of the name. Finally, if an error is found in the expression then exits with "SYNTAX ERROR"
STRTXT	DE81	Sets Y,A equal to TXTPTR plus c and fall into STRLIT
PARCHK	DEB2	Evaluate expression within paranthesis in expression.
CHKCLS	DEB8	Check at TXTPTR for a ")"
CHKOPN	DEBB	Check for "("
CHKCOM	DEBE	Check for "<"
SYNCHR	DEC0	Check at TXTPTR for character in A
PTRGET	DFE3	Read a variable name for CHRGET and find it in memory. On exit VARPNT holds its address and Y, A.If it can't find the variable, it creates it
BASIC	E000	Cold-start BASIC
BASIC2	E003	Warm-start BASIC
ISLETC	E07D	Checks to see if accumulator is a letter or number
AYINT	E10C	Make FAC an integer
GIVAYF	E2F2	Float the signed integer in A,Y
SNGFLT	E301	Float the unsigned integer in Y
ERRDIR	E306	Print "ILLEGAL DIRECT ERROR if the program is running

LABEL	ADDRESS	DESCRIPTION
STRINI	E305	Get space for creating a string and descriptor for it in DSCTMP. On entry A = string length
STRSPA	E3DD	JSR to GETSPA and store the pointer and length in DSCTMP
STRLIT	E3E7	Store a quote in ENDCHR and CHARAC so that STRLT2 will stop on it
STRLT2	E3ED	Build descriptor for string pointed at by Y,A, fall into...
PUTNEW	E42A	Some string function is returning with result in DSCTMP. Move DSCTMP to a temporary descriptor, put a pointer to the descriptor in FACMO,LO and flag result as a string
GETSPA	E452	Get space for character string. May force garbage collection On entry A contains length, returns with A unaffected an Y,X pointing to space
GARBAG	E484	Garbage collection, move all currently used strings up in memory
CAT	E597	Concatanate two strings, TXTPTR points to the '+' and FACMO,LO points at the first string whose descriptor is pointed to be STRNG1 to location pointed to by FORPNT
MOVSTR	E5E2	Move string pointed to by Y,X, length A to location pointed to FRESPA
FRESTR	E5FD	Check to see if last FAC result was a string and fall into FREFAC
FRETMP	E604	Free a temporary string, on entry pointer to descriptor in Y,X
GTBYTC	E6F5	
GETBYT	E6F8	
CONINT	E6FB	
GETNUM	E746	
COMBYTE	E74C	
GETADR	E752	Converts FAC into a 2 byte integer in LINNUM
FADDH	E7A0	Add 1/2 to FAC

LABEL	ADDRESS	DESCRIPTION
FSUB	E7A7	Move the number pointed at by Y,A to ARG then fall into...
FSUBT	E7AA	Subtract FAC from ARG
FADD	E7BE	Move the number pointed at by Y,A to ARG then fall into...
FADDT	E7C1	Add FAc to ARG
FMULT	E97F	Move the number pointed at by Y,A to ARG then fall into...
FMULTT	E982	Mulitply FAC by ARG
CONUPK	E9E3	Load ARG from memory pointed to by Y,A
MUL10	EA39	Multiply FAC by 10
DIV10	EA55	Divide FAC by 10
FDIV	EA66	Move number in memory into ARG, fall into...
FDIVT	EA69	Divide ARG by FAC
MOVFM	EAF9	Move memory into FAC
MOV2F	EB1E	Pack FAC and move it into temp register 2
MOV1F	EB21	Pack FAC and move it into temp register 1
MOVML	EB23	Pack FAC and move it to zero page
MOVXF	EB27	Store FAC into location pointed to by (FORPNT).
MOVMF	EB2B	Pack FAC and move it
MOVFA	EB53	Move ARG to FAC
MOVAF	EB63	Move FAC to ARG
SIGN	EB82	set accumulator according to the sign of FAC
SGN	EB90	Set A according to the value of FAC
FLOAT	EB93	Float accumulator
ABS	EBAF	FAC = ABS(FAC)
FCOMP	EBB2	Compare FAC with a packed number in memory
INT	EC23	Greatest integer value of FAC
INPRT	ED19	Print "IN"
LINPRT	ED24	Print current line number
PRINTFAC	ED2E	Print value of FAC
FOUT	ED34	Creates a string in FBUFFR, call strout to print it
SQR	EE8D	Square root of FAC
FPWRT	EE97	Exponentiation of ARG to the FAC power
NEGOP	EED0	FAC to -FAC

LABEL	ADDRESS	DESCRIPTION
EXP	EF09	Raise e to the FAC power
RND	EFAE	Form a random number in FAC
COS	EFEA	Cos(FAC)
SIN	EFF1	Sin(FAC)
TAN	F03A	Tan(FAC)
ATN	F09E	Arctan(FAC)
PLOTFNS	F1EC	Get 2 lo-res plotting co-ordinates
HANDLERR	F2E9	Error handling routine
RESUME	F317	Restore CURLIN and TXTPTR
HGR2	F3D4	Select the secondary hi-res page and fall through to HCLR
HGR	F3DE	Select the primary hi-res page and fall through to..
HCLR	F3EE	Clear hi-res screen to black
BKGND	F3F2	Make the hi-res screen the last colour plotted
HPOSN	F453	Call HPOSN then plot a point at the cursor's position. A = vertical position, Y,X = horizontal position
HPLOT	F453	Calls HPOSN then tries to plot a point
XDRAW	F65D	Draw the shape pointed at by Y,X by inverting the current colour. On entry A is the rotation factor
HFNS	F6B9	Get Hi-res plotting co-ordinates. Leaves the registers set up for the HPOSN
SETHCOL	F6EC	Set the hi-res colour to X,X must be less than 8
GETARYPT	F7D9	Read a variable name. It leaves LOWTR pointing to name a variable array or prints "OUT OF DATA" if not found
MONITOR		
PLOT	F800	Plot a lo-res point
HLINE	F819	Draw a lo-res horizontal line
VLINE	F828	Draw a lo-res vertical line
SCRN	F871	Find colour of lo-res point
PRNTYX	F940	Prints the contents of Y and X as hex digits
PRNTAX	F941	Print the contents of A and X as hex

LABEL	ADDRESS	DESCRIPTION
PRBLNK	F94A	Print 3 spaces
PRBL2	F94A	Print X spaces, zero equals 255 spaces
PREAD	FB1E	Read a games paddle, the X register contains the number of the number of the paddle (0 to 3) and returns a number from $0 to $FF in the Y register
TABV	FB5B	Save A in CV and jump to VTAB
BELL1	FBD9	If A contains $87 then beep speaker
VTAB	FC22	Get a vertical position of cursor, fall into...
VTABZ	FC24	Calculate vertical position of cursor
HOME	FC58	Home cursor and clear screen
WAIT	FCA8	Delay proportional to A
RDKEY	FD0C	Read a character from the current input device
KEYIN	FD1B	Read the keyboard
GETLNZ	FD67	Output a return and fall into...
GETLN	FD6A	Print prompt character and get a line of text
GETLN1	FD6F	Get a line but no prompt printed
PRBYTE	FDDA	Print A as 2 hex digits
COUT	FDED	Standard output routine
COUT1	FDF0	Print to screen

APPENDIX 4

Apple Character Set

CHARACTER	INVERSE	FLASHING	NORMAL
<space>	32	96	160
!	33	97	161
"	34	98	162
#	35	99	163
$	36	100	164
%	37	101	165
&	38	102	166
'	39	103	167
(	40	104	168
)	41	105	169
*	42	106	170
+	43	107	171
,	44	108	172
-	45	109	173
.	46	110	174
/	47	111	175
0	48	112	176
1	49	113	177
2	50	114	178
3	51	115	179
4	52	116	180
5	53	117	181
6	54	118	182
7	55	119	183
8	56	120	184
9	57	121	185
:	58	122	186
;	59	123	187
<	60	124	188
=	61	125	189
>	62	126	190
?	63	127	191

CHARACTER	INVERSE	FLASHING	CONTROL	NORMAL	LOWER CASE
@	0	64	128	192	224
A	1	65	129	193	225
B	2	66	130	194	226
C	3	67	131	195	227
D	4	68	132	196	228
E	5	69	133	197	229
F	6	70	134	198	230
G	7	71	135	199	231
H	8	72	136	200	232
I	9	73	137	201	233
J	10	74	138	202	234
K	11	75	139	203	235
L	12	76	140	204	236
M	13	77	141	205	237
N	14	78	142	206	238
O	15	79	143	207	239
P	16	80	144	208	240
Q	17	81	145	209	241
R	18	82	146	210	242
S	19	83	147	211	243
T	20	84	148	212	244
U	21	85	149	213	245
V	22	86	150	214	246
W	23	87	151	215	247
X	24	88	152	216	248
Y	25	89	153	217	249
Z	26	90	154	218	250
[	27	91	155	219	251
	28	92	156	220	252
]	29	93	157	221	253
	30	94	158	222	254
	31	95	159	223	255

APPENDIX 5

Table 1

Hexadecimal to Decimal Conversion Table

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

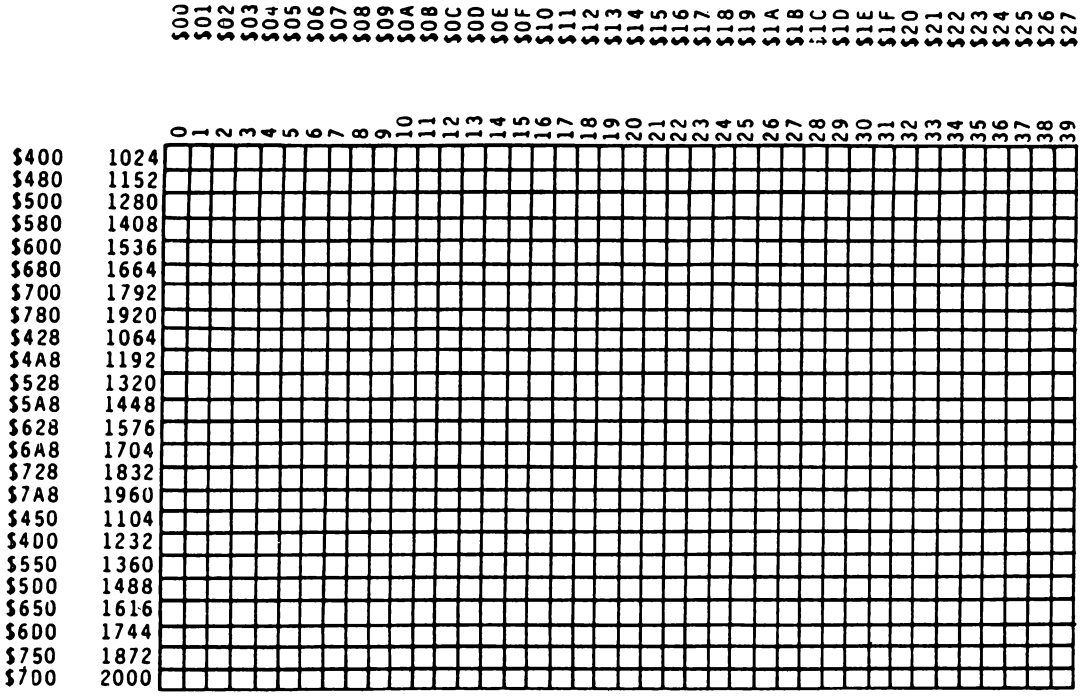
Table 2

ASCII Character Set

HEX LSB	MSB BIN	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SPACE	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	-	o	DEL

APPENDIX 6

Low-Resolution Screen Map



APPENDIX 7

Different Varieties of Apple

The Apple][c is the latest revision to the Apple][since it was first introduced in 1977. This book was primarily written for the Apple IIe.

If you wish to write a machine code program that will work on previous versions of the Apple you need to know the differences between them so the program does not call a routine that either does not exist on earlier machines or does something different. These changes are mainly limited to the monitor and enhancements to the soft switches.

The original Apple][had Integer BASIC in ROM and the so called 'monitor ROM' monitor sometimes refered to as the F8 monitor which included the step and trace commands for helping you to trace the execution of machine code program.

Next came the Apple][+ which has Applesoft in ROM and an altered monitor called the 'autostart ROM' because when the Apple was turned on, it would automatically boot the disk drive which the previous version would not, it would put you straight into the monitor and you had to boot the disk manually.

Now with the //e version the monitor has been changed again but it still retains the same version of BASIC as the][+, bugs and all.

As much compatability has been kept between all three versions of the monitor as possible so if you wish to make sure your program is transportable, at least as far as the monitor routines are concerned, then refer to Appendix 3.

Any program using CALLs to Applesoft will work on both the //e and the][+, they may work on Apple][s if they have a RAM card with Applesoft loaded onto it or a ROM card and it has been selected.

An easier problem is DOS compatiability. The same version of DOS will run on any variety of Apple. It was only slightly patched for the //e.

The only slight concern is for someone using DOS 3.2 plus or even older versions but this must now represent a very small proportion of users.

The Difference between American and European Apples

While American and European Apples are functionally identical, that is, any program that runs on an American machine will run on a European machine and vice versa there are differences in clock speed and in the displays.

Due to the differences in the display standards (NTSC in America and PAL in Europe) European Apples run slightly slower than American ones, the difference is 0.995 MHz (MegaHertz) compared to 1.023 MHz.

Another difference is in terminology. The PAL (Programmed Array Logic) is known as a ULA (Uncommitted Logic Array) in Britain.

INDEX

7120-5

Apple® Assembly Language Programming

Malcolm Whapshott

This complete course in assembly language programming provides a self-paced, structured learning experience matched with a full-featured assembler on the enclosed software.

Guides the beginner step by step through all the essentials of assembly language programming: branching, screen display, addressing modes, interrupts, macro instructions, floating point calculations, and using the built-in subroutines of your machine.

The *Software* contains a full-featured assembler complete with labels, memory labels, macros, and more. *Plus* a binary/hexadecimal conversion tutor.

The *Book* contains carefully sequenced instruction in assembly language programming, detailed explanations, and exercises with their solutions.

Requires: Apple II (with Applesoft), Apple II Plus, Apple IIe, or Apple IIc, 1 disk drive, and a monitor.

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

ISBN 0-8104-7120-5

Cover Photo: David Wagner